From PyTorch to Calyx: An Open-Source Compiler Toolchain for ML Accelerators

Jiahan Xie
Computer Science and Engineering Department
UC Santa Cruz
Santa Cruz, USA
jxie84@ucsc.edu

Evan Williams
Computer Science Department
Cornell University
Ithaca, USA
emw236@cornell.edu

Adrian Sampson
Computer Science Department
Cornell University
Ithaca, USA
asampson@cs.cornell.edu

***Abstract*—We present an end-to-end open-source compiler toolchain that targets synthesizable SystemVerilog from ML models written in PyTorch. Our toolchain leverages the accelerator design language Allo, the hardware intermediate representation Calyx, and the CIRCT project under LLVM. We also implement a set of compiler passes for memory partitioning, enabling effective parallelism in memory-intensive ML workloads. Experimental results demonstrate that our compiler can effectively generate optimized FPGA-implementable hardware designs that perform reasonably well against closed-source industry-grade tools such as Vitis HLS.**

***Index Terms*—Compiler, Open-Source, Hardware Accelerators, Machine Learning**

I. Introduction

As ML applications grow in complexity [2] and performance demand, general-purpose processors are no longer optimal for ML workloads due to their limited efficiency and high energy overhead in computation-intensive tasks [1]. Although custom hardware accelerators [3] offer significant performance and energy efficiency gains [4], designing them at the register-transfer level (RTL) is difficult. Hardware description languages (HDLs) such as SystemVerilog or VHDL offer fine-grained control over circuit behavior, but writing and verifying correct HDL code is time-consuming and error-prone. With the rapid evolution of ML models and the explosive need for custom hardware to run modern workloads, hardware design cycles must become faster and more adaptable.

High-level synthesis (HLS) offers a promising alternative for designing ML accelerators [8] [9] [16]. HLS allows us to compile high-level functional specifications from software (typically written in C or C++) to synthesizable RTL suitable for hardware implementation. However, few HLS approaches exist for compiling directly from the languages and frameworks that are commonly used to express ML models, such as PyTorch. Moreover, the prior work relies heavily on closed-source commercial HLS toolchains, such as Xilinx Vitis HLS [17]. Recent MLIR-based HLS systems demonstrate the potential for open-source compiler stacks for accelerator design [14].

Therefore, we introduce an end-to-end open-source compiler toolchain that generates synthesizable SystemVerilog from PyTorch models through a structured compilation flow. Our system uses Allo [7] to translate PyTorch programs into MLIR, leverages domain-specific MLIR dialects [11] to preserve high-level tensor structure, and relies on the CIRCT [6], [12] infrastructure to perform hardware lowering. The resulting program is expressed in Calyx, whose explicit separation of control and hardware structure allows us to encode accelerator architectures and optimizations cleanly. Finally, we compile Calyx to SystemVerilog and use standard FPGA vendor tools such as Xilinx Vivado to synthesize, place–and–route, and deploy accelerators. Our contributions are as follows:

- An end-to-end open-source compiler stack from PyTorch to synthesizable SystemVerilog using Allo, CIRCT, and Calyx.
- Memory banking and partitioning analyses in Calyx enabling safe and efficient parallel access patterns.
- FPGA evaluation showing performance up to $2.21\times$ faster than Vitis HLS.

II. Background and Challenges

A. Allo

Allo [7] is a compiler for constructing large-scale, high-performance hardware accelerators. Its lowering pipeline compiles PyTorch programs into MLIR while preserving tensor semantics, control flow, and data-layout information. The resulting structured MLIR program integrates cleanly with downstream dialects and compiler infrastructures, forming a bridge between high-level ML frameworks and hardware generation backends.

B. Calyx

Calyx [5], [13] is an intermediate representation (IR) and compiler infrastructure designed for generating hardware accelerators from high-level programming languages. It explicitly separates control flow from structural hardware descriptions, enabling optimizations that leverage both perspectives. The control sublanguage expresses imperative constructs such as loops and conditionals, while the structural sublanguage instantiates hardware components and defines the wiring between them. This split representation allows compiler frontends to describe both computation and architecture naturally.

The Calyx compiler then lowers the program into synthesizable RTL through its series of transformation and optimization passes.

C. CIRCT

Circuit IR Compilers and Tools (CIRCT) [6] is an open-source, LLVM-based infrastructure for building hardware compilers. Built atop MLIR (Multi-Level Intermediate Representation), CIRCT provides a suite of hardware-specific dialects and transformation passes to support the development of custom compilation flows, hardware synthesis pipelines, and intermediate representations. Notable dialects include `HW` (hardware module descriptions), `FSM` (finite-state machines), and various dialects for scheduling and memory management.

CIRCT aims to accelerate compiler development for hardware by offering a modular and extensible framework [12]. Calyx is integrated into CIRCT as one of its dialects, allowing users to implement transformation passes that lower high-level MLIR dialects (e.g., `SCF`) to Calyx. This integration enables full-stack compilation from high-level MLIR programs to hardware-level IRs, which can then be lowered into synthesizable SystemVerilog.

D. Challenges

Despite these advances, several challenges remain in building an end-to-end compiler stack for ML workloads targeting hardware accelerators:

Bridging the software-hardware gap. Although Calyx unifies software-style control and hardware-style structure, programming directly in Calyx remains low-level and resembles writing HDL code. For ML workloads primarily written in Python, an end-to-end flow is needed that compiles high-level Python programs to CIRCT dialects and ultimately to synthesizable hardware. Bridging this gap between Python and hardware remains a central challenge.

Floating-point support. Floating-point arithmetic is a core component of modern ML models but is notoriously difficult to implement efficiently in hardware. Unlike integer arithmetic, floating-point operations require managing special cases such as `NaNs` and infinities, and involve more complex circuits. The CIRCT and Calyx ecosystems originally lacked floating-point support due to this complexity. However, ML workloads depend heavily on floating-point math, making it essential to extend these infrastructures with robust support for floating-point constants and operations.

Limited coverage of ML kernels. The early Calyx-based CIRCT flow [11] handled only a narrow subset of SCF constructs, restricting its applicability to simple kernels such as matrix multiplication. It lacked support for richer ML components, including multi-layer perceptrons, nonlinear activations, and softmax operators used in attention. Moreover, ML models rely on parameterized storage - weights and biases - which requires explicit modeling of data layout and memory organization. Supporting multi-module models and inter-function orchestration also adds complexity to the compiler pipeline.

Performance optimization through parallelism. Once a functional compiler flow is established, the next goal is improving hardware performance. Parallel execution is key. Calyx supports parallelism as a first-class control construct, translating concurrent execution into multiple finite-state machines (FSMs) while checking for resource contention, such as memory port conflicts. Due to the memory-intensive nature of ML workloads, achieving correct and efficient parallel execution requires sophisticated compiler analyses and transformations to avoid hazards and resource contention at runtime.

III. Technical Contribution

A. Overview

The goal of this work is to execute ML programs written in PyTorch on custom hardware accelerators. We use FPGAs as the prototyping platform and develop a fully open-source compilation pipeline that transforms high-level Python programs into synthesizable hardware designs.

Our toolchain is composed of several open-source components. We begin by using the Allo framework [7] to compile PyTorch models into MLIR programs. These MLIR programs are then progressively lowered via native MLIR passes to dialects supported by CIRCT, leveraging CIRCT's integration with MLIR to apply software-like optimizations and analysis. While CIRCT brings the program close to hardware form, Calyx provides explicit control and memory structure needed for accelerator-oriented analysis and transformations.

Once the program reaches a form compatible with Calyx, we emit code in the Calyx intermediate representation. The Calyx compiler then performs hardware-specific transformations and generates synthesizable SystemVerilog. This hardware design is finally deployed to an FPGA, completing the software-to-hardware compilation path.

B. Lowering to Calyx

To generate synthesizable hardware designs from high-level MLIR programs, the first step is to lower operations from high-level software-oriented dialects to the hardware-centric Calyx dialect. This requires bridging the semantic gap between software abstractions such as control flow, function calls, and floating-point operations, in addition to their hardware realizations. Prior work by Urbach and Petersen [11] established an initial foundation for this lowering, which we extend and build upon in our compiler.

Figure 1 shows the compilation flow we developed and orchestrated to lower PyTorch to Calyx. We leverage Allo to lower to the MLIR Linalg dialect. We then lower to the MLIR Affine dialect, and then the Memref and SCF dialects. We use CIRCT to produce Calyx, and Vivado HLS to execute the design on an FPGA.

CIRCT lowers structured control flow constructs (e.g., `for`, `while`, `if`) into Calyx by constructing explicit state machines and scheduling logic. Loops are lowered using dedicated registers for induction variables, along with Calyx control operators like `repeat`, `seq`, and `while`. Conditional branches are translated to `if` operations with value-passing handled through

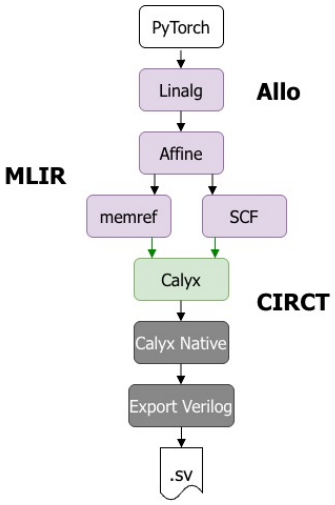

Fig. 1. Compilation pipeline from PyTorch through Allo to Calyx.

auxiliary result registers. A hierarchical control schedule is constructed by traversing the program's control flow graph, resulting in a Calyx control block that faithfully captures the structure of the original software logic.

Function boundaries are lowered into Calyx components. Each software-level function becomes a hardware module with scalar arguments mapped to input/output ports and memory arguments instantiated as memory components with structured interfaces. Function calls are translated into component instantiations and invocations, enabling modular and reusable hardware. When a top-level function contains memory references or internal allocations, an additional wrapper component is generated to externalize memory and expose it at the hardware boundary.

We developed a full, general floating-point library for Calyx to provide floating-point support in CIRCT, including the integration of the Berkeley HardFloat [15] components. Floating-point support is modeled at the bit-level, since CIRCT and Calyx fundamentally operate on integer-typed bitvectors. To handle floating-point constants, both ordinary and special values are represented using IEEE-754 encodings as bitvectors. To retain readability and support debugging, decimal representations are attached as attributes, enabling CIRCT and Calyx to internally convert between human-readable and bit-level formats. All floating-point constants are thus treated as raw bitpatterns during lowering. The Calyx native compiler integrates HardFloat modules to implement floating-point operations, generates the necessary hardware components, and preserves the original MLIR semantics during lowering.

Altogether, this lowering process forms the foundation of our end-to-end compiler stack, allowing software-level MLIR programs to be expressed in the Calyx IR and synthesized into RTL.

C. Memory and Parallelism Optimizations

The primary optimization goal in our compilation flow is to reduce the wall-clock latency of the forward pass of ML models. To achieve this, we leverage parallelism to increase hardware throughput. Calyx supports parallel execution as a first-class control construct, allowing us to explicitly model concurrent computation. Our task is to expose and maximize parallelism in memory access patterns while adhering to hardware constraints.

In hardware, there are two main types of parallelism: pipeline parallelism, which breaks a task into stages using different resources, and data parallelism, which duplicates hardware units to perform computations concurrently. We adopt the latter and focus on memory partitioning (also known as memory banking), since Calyx assumes that each memory unit supports at most one read or write per cycle. To support concurrent accesses, we duplicate memories and route operations to different banks.

Static optimization of memory concurrency is challenging. Calyx detects access violations during simulation, but our goal is to eliminate such contention through static analysis and code transformations. However, memory banking introduces complexity because it increases the number of memory ports and leads to control structures that select which bank to access, often with nested conditionals. These extra control paths increase both latency and resource usage if left unoptimized.

Our implementation supports cyclic memory partitioning, and we assume this scheme throughout. To route accesses to the correct bank, we use a `switch` statement (or nested `if-else` chains where `switch` is unavailable). A naive implementation that directly emits control branches for each bank leads to code-size blow-up. For a memory with d dimensions and a partition factor c, the number of unique control branches scales as c^d. In Calyx, these branches are all instantiated as hardware, even if only one is active at runtime. This not only increases area usage but also results in deeper control FSMs, which hurt performance.

Further complications arise when attempting to parallelize memory access. Calyx's `par` construct does not perform constant folding, even mutually exclusive guarded operations, such as executing one statement when a flag is statically known to be true and a different statement when it is false, are still instantiated as parallel hardware. Consequently, although only one branch can execute semantically, both are present in the generated circuit and may contend for shared resources, leading to unintended memory conflicts.

To illustrate the problem concretely, consider the simple loop in Listing 1, which writes to a four-element memory. Suppose we apply cyclic memory banking with a factor of 2, resulting in two memory banks, `mem_bank_0` and `mem_bank_1`. A straightforward transformation produces the code shown in Listing 2.

To parallelize this loop, we materialize it with a factor of 2 using nested `seq` and `par`, as shown in Listing 3. However, each `par` block must be unrolled into separate arms with

```
for (int i = 0; i < 4; ++i) {
   mem[i] = i;
}
```

Listing 1. Original loop

```
for (int i = 0; i < 4; ++i) {
   if (i % 2 == 0) {
      mem_bank_0[i / 2] = i;
   } else {
      mem_bank_1[i / 2] = i;
   }
}
```

Listing 2. Naive memory banking

```
seq for (int i = 0; i < 2; ++i) {
   par for (int j = 0; j < 2; ++j) {
      int new_index = 2 * i + j;
      if (new_index % 2 == 0) {
         mem_bank_0[new_index / 2] = new_index;
      } else {
         mem_bank_1[new_index / 2] = new_index;
      }
   }
}
```

Listing 3. Materialized loop with banking

```
seq for (int i = 0; i < 2; ++i) {
   parallel execution {
      execute par-arm-0 {
        int new_index = 2 * i + 0;
        if (new_index % 2 == 0) {
          mem_bank_0[new_index / 2] = new_index;
        } else {
          mem_bank_1[new_index / 2] = new_index;
        }
      }
      execute par-arm-1 {
        int new_index = 2 * i + 1;
        if (new_index % 2 == 0) {
          mem_bank_0[new_index / 2] = new_index;
        } else {
          mem_bank_1[new_index / 2] = new_index;
        }
      }
   }
}
```

Listing 4. Unrolled parallel execution

statically known indices. The resulting unrolled version is shown in Listing 4.

Although each arm only triggers one branch of the `if-else`, both branches are instantiated. If two arms write to the same memory bank due to symbolic indices, a write conflict will occur at runtime. Since Calyx lacks symbolic constant folding in this context, the compiler cannot resolve the access pattern statically.

We implement two techniques to ensure safe and efficient memory parallelism: (1) we express banking by embedding the bank index into the memory's dimensional layout instead of guarding accesses with conditional logic; and (2) we rewrite loop nests whose structure would otherwise duplicate sequential controllers when executed in parallel.

```
seq for (int i = 0; i < 2; ++i) {
   par for (int k = 0; k < 2; ++k) {
      mem[k][i] = 2 * i + k;
   }
}
```

Listing 5. Bank dimension encoding

```
seq for (int i = 0; i < 2; ++i) {
   parallel execution {
      execute par-arm-0 {
         mem[0][i] = 2 * i + 0;
      }
      execute par-arm-1 {
         mem[1][i] = 2 * i + 1;
      }
   }
}
```

Listing 6. Conflict-free parallel write

For the first, rather than emitting branch logic per access, we raise the memory's dimensionality and bake the bank index into the first dimension. An example of this transformation is shown in Listing 5.

Unrolling this loop yields the conflict-free parallel write shown in Listing 6.

Here, the bank index is a compile-time constant in each parallel arm, ensuring that memory accesses are disjoint and contention-free. This approach enables us to preserve parallelism without incurring the cost of unnecessary control overhead.

The second transformation operates at the level of loop structure. This example also highlights a broader point: loop transformations that are semantically equivalent in software do not necessarily yield equivalent hardware. Consider two nestings: one where `seq(i)` surrounds `par(j)`, and another where `par(j)` surrounds `seq(i)`. Although they are semantically equivalent to `par(j)` around `seq(i)` in software, they behave very differently in hardware. In the first form, there is a single sequential controller that iterates over `i`, and each iteration triggers a parallel group over `j`. In the second form, however, each parallel arm receives its own private sequential controller for iterating over `i`, effectively duplicating the entire FSM. This replication inflates the hardware area and increases control overhead. To prevent such unnecessary duplication, our compiler detects these patterns and rewrites parallel–sequential loop nests into schedules that share control logic while preserving the intended parallelism.

Through these memory and loop-aware transformations, our compiler produces parallel Calyx programs that are both correct and efficient for hardware execution.

IV. Results and Evaluation

In this section, we evaluate our Calyx-based flow against Vitis HLS under two configurations. In Section IV-C, we compare baseline designs with no data parallelism: neither

toolchain applies any banking strategy. In Section IV-D, we enable memory banking for both flows using matched partitioning factors, schemes, and dimensions, allowing a direct comparison of parallelized configurations.

Before presenting quantitative results, it is important to contextualize the comparison. Vitis HLS has been developed and tuned for over a decade, and applies many mature and often implicit optimizations — including automatic scheduling, resource sharing — that are not present in our current Calyx-based flow and cannot be selectively disabled or inspected through `pragma`s. For this reason, we do not expect Calyx to outperform Vitis HLS in baseline latency or resource usage. Rather, the interesting question is how competitive Calyx can be despite its simpler optimization stack, and whether targeted compiler transformations can substantially close the gap. Our results demonstrate that, although Vitis retains an advantage in baseline configurations, Calyx approaches competitive performance under parallelized banking, while remaining fully open-source and compiler-controlled. We view this as a promising foundation for continued work in bringing Calyx-based compilation closer to state-of-the-art commercial HLS flows.

A. Experimental Setup

All experiments are conducted on a dual-socket Intel Xeon Gold 6230 workstation (20 cores, 40 SMT threads per socket at 2.10,GHz) with 512 GB RAM. The FPGA accelerator platform we are targeting is a Xilinx Alveo U50 board. For the commercial toolchain, we use AMD/Xilinx Vitis HLS and Vivado version 2023.2.

For all Calyx designs, we report post–place-and-route operating frequencies obtained from Vivado. For Calyx-generated RTL, we synthesize and place-and-route the design in Vivado 2023.2, sweeping the timing constraint upward until the design produces zero or positive worst-negative slack (WNS). The highest frequency meeting timing is then used to convert cycle counts into wall-clock latency. Cycle counts are obtained by simulating the SystemVerilog code with Verilator.

Vitis HLS directly reports latency in milliseconds. We similarly sweep the timing constraint for each Vitis design in Vivado 2023.2 and select the implementation that achieves the lowest reported latency without violating WNS.

As a result, the two toolchains' latency numbers are obtained via different methodologies: Vitis provides an estimated latency based on its internal scheduling and performance model timing, while our Calyx flow reports simulated dynamic behavior under the achieved frequency. This mismatch introduces an inherent margin of error in absolute comparisons, particularly when the reported latencies are close. However, we apply the same timing-constraint sweep procedure to both flows and use each toolchain's standard reporting method, and we primarily interpret the results in terms of relative trends across models and partitioning factors rather than claiming exact absolute latency equivalence.

B. Benchmark Models

We evaluate the performance and resource usage of our compiler by benchmarking three representative machine learning models and comparing them against a commercial HLS toolchain, Xilinx Vitis HLS. The models include a feed-forward neural network (FFNN), a convolutional neural network (CNN), and a multi-head attention (MHA) module.

The FFNN model takes an input of 64 features, followed by a fully connected layer of size 64×48, a ReLU activation, and a second fully connected layer of size 48×4.

The CNN model processes a 80×60 color image with 3 channels. The first layer performs a 2D convolution with 5×5 kernels, 3 input channels, and 8 output channels using unit strides. This is followed by a ReLU activation and a max-pooling layer with a 2×3 window. The resulting feature map is flattened and passed through a fully connected layer for binary classification.

The MHA model is derived from the Transformer architecture and uses 2 attention heads. Each head operates on a 21-dimensional subspace of a 42-dimensional embedding, with causal masking for autoregressive decoding.

C. Comparison Across Models

For comparison, we use Vitis HLS, a widely used commercial HLS tool. The Allo project provides a shared frontend that lowers PyTorch models to MLIR, which is then further compiled to either HLS C++ for Vitis or Calyx for our flow. To ensure fairness, the MLIR input is held constant across both paths. All HLS pragmas are disabled except for `#pragma ARRAY_PARTITION`, which is used to match our memory banking configuration. We ensure consistent banking factors, schemes (cyclic), and dimensions across both flows. Vitis's automatic scheduling and optimization passes are left on, as they cannot be turned off.

Figure 2 shows the wall-clock latency of each model across the two toolchains. Vitis outperforms our Calyx-based in most cases. This performance gap is largely due to limitations in Calyx's memory model: Calyx only supports single-dimensional memories, requiring us to flatten multi-dimensional tensors. As a result, access indices—often affine expressions of loop variables—are lowered to hardware as expensive arithmetic operations like multiplication and modulo, which introduce latency. Furthermore, the CNN model's convolution and pooling layers naturally lead to deeper loop nests, amplifying this cost. Without common CNN optimizations such as line buffering, the performance gap is further exposed. However, with the addition of our memory banking strategy, we find that Calyx achieves a $2.21\times$ speedup over Vitis.

Table I and Table II report resource utilization across the same models. We observe that Vitis uses more BRAMs for storing weights and biases, while Calyx consumes significantly more LUTs and FFs due to its use of explicit finite state machines (FSMs) for control. In several cases, Calyx exhibits a 3–4$\times$ increase in LUT usage relative to Vitis, representing a substantial area overhead. This overhead stems primarily from Calyx's explicit and unoptimized control representation, in which structured software-level control flow is lowered into hardware FSMs without aggressive scheduling, resource

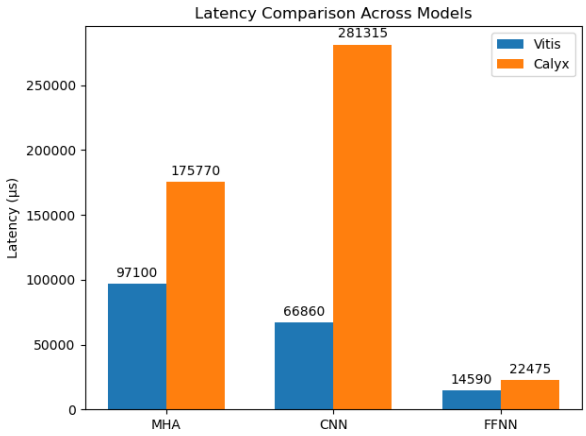

Fig. 2. Wall-clock latency comparison across models.

TABLE I
LUT AND FF USAGE ACROSS MODELS.

Model	LUTs		FFs	
	Vitis	Calyx	Vitis	Calyx
MHA	7846	33312	4017	5561
CNN	3136	4574	1815	1223
FFNN	2011	3730	1281	742

TABLE II
BRAM AND DSP USAGE ACROSS MODELS.

Model	BRAMs		DSPs	
	Vitis	Calyx	Vitis	Calyx
MHA	194	71	19	67
CNN	213	43	5	14
FFNN	43	9	5	6

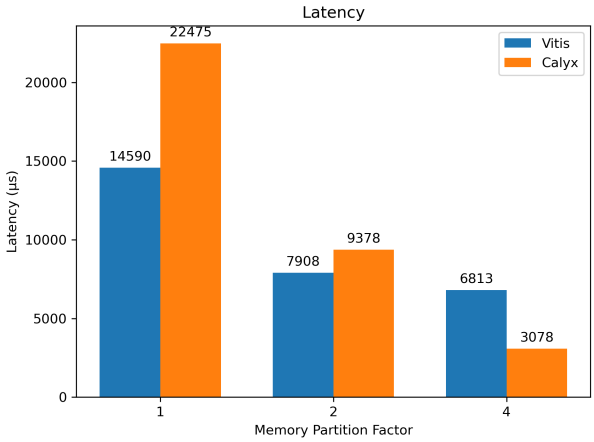

Fig. 3. Latency vs. partition factor for FFNN.

sharing, or control minimization. This area overhead is therefore a current limitation of our approach and represents a barrier to practical industrial adoption. Accordingly, we view the present system as a research and prototyping tool rather than a replacement for mature commercial HLS compilers, and reducing control overhead while preserving analyzability is an important direction for future work.

D. Memory Banking for Parallelism

To further investigate the impact of memory banking, we conduct a detailed case study on the FFNN model. In this study, every memory is partitioned cyclically along each dimension. In Vitis, this is done via `#pragma ARRAY_PARTITION`, while in Calyx the partitioning is implemented via our compiler pass. We compare both latency and resource usage across varying banking factors.

Figure 3 shows the latency across different partition factors.

For `factor=1` and `factor=2`, Vitis performs better. However, at `factor=4`, Calyx becomes faster. Moreover, the relative speedup for Vitis is modest: increasing the banking factor from 2 to 4 yields diminishing returns, with a speedup of only $7908/6813 \approx 1.16$. Given that all matrices are two-dimensional, the theoretical maximum speedup is $2^2 = 4$ under ideal conditions. The limited gain suggests that other bottlenecks remain.

In contrast, the Calyx-based flow shows significant improvement with higher partitioning. The speedup from `factor=1` to `factor=2` is $22475/9378 \approx 2.40$, and from `factor=2` to `factor=4` is $9378/3078 \approx 3.05$. This demonstrates the effectiveness of our memory banking analysis and transformation passes, which allow Calyx to unlock more parallelism at the memory level.

Table III and Table IV show the resource usage corresponding to these experiments. DSP usage is comparable across both toolchains. Vitis uses slightly more BRAMs and FFs, while Calyx consumes significantly more LUTs, particularly at `factor=4`, due to the additional control logic needed for managing multiple memory banks. These results illustrate the tradeoff between performance and hardware complexity introduced by fine-grained memory partitioning.

V. FUTURE WORK

There are several promising directions for extending this compiler framework. First, we plan to broaden the set of supported ML kernels by adding lowering patterns and optimizations for additional operators beyond the models evaluated in this work. Expanding kernel coverage will further improve the applicability of the toolchain to real-world workloads.

Further, we aim to extend the compilation flow to support the backward pass. This requires lowering differentiation-related operations, handling gradient propagation, and managing the additional memory and parallelism introduced by training workloads. Supporting training as well as inference with the same flow would enable end-to-end hardware generation, allowing accelerator designs to perform on-device learning, fine-tuning, and inference without relying on external frameworks.

TABLE III
RESOURCE USAGE VS. PARTITION FACTOR FOR THE FFNN MODEL.

Partition factor	LUTs		FFs	
	Vitis	Calyx	Vitis	Calyx
1	2011	3730	1281	742
2	6021	13197	4036	3145
4	13799	49121	15083	10657

TABLE IV
MEMORY AND DSP USAGE VS. PARTITION FACTOR FOR THE FFNN MODEL.

Partition factor	BRAMs		DSPs	
	Vitis	Calyx	Vitis	Calyx
1	43	9	5	6
2	39	10	7	20
4	64	20	80	69

As discussed in Section III-C, hardware parallelism includes both data parallelism and pipeline parallelism. An important next step is to introduce pipelining support analogous to Vitis' `#pragma HLS PIPELINE`. This entails generating pipelined control schedules using the scheduling and loop-pipelining dialects in CIRCT and automatically identifying program regions that benefit from initiation-interval optimization. Adding first-class pipelining capabilities will allow our compiler to more effectively exploit fine-grained parallelism and reduce overall latency.

A further important direction is reducing the substantial control and logic overhead currently introduced by our lowering strategy, as shown in Section IV-C.

Beyond further optimizations analogous to Vitis HLS pragmas, we are also currently exploring alternate compilation paths through the MLIR/CIRCT infrastructure. By integrating with tools such as Torch-MLIR [18] and [19], we can make our open-source toolchain more accessible and provide more options to the end-user.

VI. CONCLUSION

This work presents a complete open-source compilation toolchain that transforms PyTorch programs into synthesizable hardware designs using Allo, CIRCT, and Calyx. We bridge the gap between software-level ML programs and hardware accelerators by implementing support for structured control flow, function modeling, floating-point arithmetic, and memory layout transformations.

We focus in particular on optimizing memory access concurrency through static memory banking and control restructuring. Our evaluation on three representative ML models shows that while the Calyx-based flow trails behind commercial tools like Vitis in general-purpose scheduling and resource efficiency, it demonstrates promising performance gains when aggressive memory partitioning is applied. These results highlight the potential of Calyx as a research and prototyping platform for hardware accelerator compilation, especially in settings where open-source and customization are prioritized.

REFERENCES

[1] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proc. 37th Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 37–47, doi: 10.1145/1815961.1815968.

[2] P. Villalobos, J. Sevilla, T. Besiroglu, L. Heim, A. Ho, and M. Hobbhahn, "Machine learning model sizes and the parameter gap," *arXiv preprint* arXiv:2207.02852, 2022. [Online]. Available: https://arxiv.org/abs/2207.02852.

[3] D. Ali, A. U. Rehman, and F. H. Khan, "Hardware accelerators and accelerators for machine learning," in *Proc. Int. Conf. IT Ind. Technol. (ICIT)*, 2022, pp. 1–7, doi: 10.1109/ICIT56493.2022.9989124.

[4] M. Vaithianathan, M. Patil, S. Ng, and S. Udkar, "Comparative study of FPGA and GPU for high-performance computing and AI," *Int. J. Adv. Comput. Technol.*, vol. 1, pp. 37–46, 2023, doi: 10.56472/25838628/IJACT-V1I1P107.

[5] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *Proc. ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2021, pp. 804–817, doi: 10.1145/3445814.3446712.

[6] LLVM CIRCT Developers, "CIRCT: Circuit IR compilers and tools," 2024. [Online]. Available: https://github.com/llvm/circt.

[7] H. Chen, N. Zhang, S. Xiang, Z. Zeng, M. Dai, and Z. Zhang, "Allo: A programming model for composable accelerator design," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, pp. 171:1–171:28, 2024, doi: 10.1145/3656401.

[8] S. I. Venieris and C. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *Proc. IEEE Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2016, pp. 40–47, doi: 10.1109/FCCM.2016.22.

[9] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2016, Art. no. 17, pp. 1–12.

[10] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, 2021, pp. 2–14, doi: 10.1109/CGO51591.2021.9370308.

[11] M. Urbach and M. B. Petersen, "HLS from PyTorch to SystemVerilog with MLIR and CIRCT," presented at the 2nd Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'22), 2022. [Online]. Available: https://capra.cs.cornell.edu/latte22/paper/2.pdf.

[12] J. Demme and A. Landy, "Using CIRCT for FPGA physical design," presented at the 2nd Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'22), 2022. [Online]. Available: https://capra.cs.cornell.edu/latte22/paper/10.pdf.

[13] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, "Predictable accelerator design with time-sensitive affine types," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*, 2020, pp. 393–407, doi: 10.1145/3385412.3385974.

[14] H. Ye, H. Jun, H. Jeong, S. Neuendorffer, and D. Chen, "ScaleHLS: A scalable high-level synthesis framework with multi-level transformations and optimizations," in *Proc. ACM/IEEE Des. Autom. Conf. (DAC)*, 2022, pp. 1355–1358, doi: 10.1145/3489517.3530631.

[15] J. R. Hauser, "Berkeley HardFloat," 2019. [Online]. Available: https://github.com/ucb-bar/berkeley-hardfloat

[16] J. Duarte *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *J. Instrum.*, vol. 13, no. 07, p. P07027, 2018, doi: 10.1088/1748-0221/13/07/P07027.

[17] AMD/Xilinx, "Vitis High-Level Synthesis User Guide," Version 2023.2, 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls.

[18] LLVM/torch-mlir contributors, "llvm/torch-mlir: A PyTorch front-end for MLIR," GitHub repository, 2025. [Online]. Available: https://github.com/llvm/torch-mlir

[19] IREE Developers, "IREE: A unified compiler and runtime for ML models built on MLIR," GitHub repository, 2025. [Online]. Available: https://github.com/openxla/iree