DESIGNING MACHINE LEARNING ACCELERATORS VIA HIGH-LEVEL SYNTHESIS THROUGH CALYX

Jiahan Xie
Department of Computer Science
Cornell University
Ithaca, NY, USA
`jx353@cornell.edu`

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

October 9, 2025

ABSTRACT

As applications grow in complexity and performance demand, general-purpose processors are no longer optimal for many domains due to their limited efficiency and high energy overhead in computation-intensive tasks [1]. This is particularly evident in the field of machine learning (ML), where model sizes and usage have grown exponentially [2], making CPUs ill-suited for inference and training workloads. Although custom hardware accelerators offer significant performance and energy efficiency gains, designing them at the register-transfer level (RTL) is time-consuming and error-prone. Furthermore, the vast majority of ML programs are written in Python, a high-level language that is far removed from RTL design.

To bridge this semantic gap, we present a complete open-source compiler toolchain that translates ML models written in Python into synthesizable SystemVerilog, targeting FPGAs as the hardware backend. Our toolchain leverages Calyx, a structured intermediate representation (IR) designed for hardware accelerators, and is integrated into the CIRCT project for extensibility and analysis. In addition to building the end-to-end flow, we design and 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 hardware from high-level ML models and achieve performance gains through static memory banking optimizations.

1 Introduction

Machine learning (ML) has become ubiquitous in today's rapidly evolving technological landscape, with models such as ChatGPT exemplifying its widespread adoption. The size and complexity of ML models are growing exponentially [2], placing increasing demands on hardware execution platforms. Traditional CPUs are ill-suited for this workload, as they are optimized for general-purpose tasks and struggle to handle the compute- and memory-intensive operations—such as matrix multiplications and nonlinear activations—common in ML workloads. Furthermore, CPUs consume significant energy simply supplying and decoding instructions [1], rather than performing the actual computations.

In contrast, hardware accelerators such as GPUs, TPUs, and custom ASICs can execute ML workloads more efficiently due to their specialized architectures [3]. Companies like NVIDIA have thrived by pairing powerful hardware with a robust software ecosystem. However, even these accelerators face challenges—especially in energy efficiency in inference [4]. This motivates the need for designing custom accelerators that deliver both high performance and low power consumption.

Despite their benefits, designing hardware accelerators remains difficult. Hardware description languages (HDLs) such as Verilog 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, hardware design cycles must become faster and more adaptable. Ideally, developers should be able to write ML models in a high-level, expressive language that is easy to reason about and optimize, and then compile these models into hardware automatically.

High-Level Synthesis (HLS) offers a promising solution. HLS compiles high-level functional specifications - typically written in C or C++ - into synthesizable RTL suitable for hardware implementation. However, this traditional HLS approach is limited. C/C++ was not designed with hardware semantics in mind, and lacks first-class constructs for parallelism, memory hierarchies, and control flow synchronization. Moreover, while Python is the dominant language in the ML ecosystem, most HLS tools do not support high-level ML frameworks like PyTorch, limiting their ability to capture domain-specific semantics. Bridging this semantic and tooling gap requires new compiler infrastructures that are hardware-aware and extensible, while also supporting high-level software-style programming.

Additionally, most commercial HLS tools, such as Xilinx Vitis, are proprietary. This creates barriers for innovation and limits the accessibility of hardware design. Open-source alternatives are crucial to democratize accelerator development and foster community-driven improvements.

In this thesis, we address these challenges by leveraging the Calyx project [5], a structured intermediate representation (IR) designed specifically for compiling high-level programs to hardware accelerators. Calyx cleanly separates control logic from hardware structure and is integrated as a dialect within the CIRCT project [6], an MLIR-based framework for hardware compiler development. We build a complete open-source toolchain that compiles ML models written in Python to MLIR, lowers them through CIRCT to Calyx, and generates synthesizable SystemVerilog for execution on FPGAs. In addition to constructing this end-to-end flow, we contribute novel compiler passes for memory partitioning optimizations, enabling parallel execution of memory-intensive workloads.

2 Background and Challenges

2.1 Calyx

Calyx 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 a series of transformation and optimization passes.

2.2 CIRCT

Circuit IR Compilers and Tools (CIRCT) 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 `SCF` (structured control flow), `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. 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.

2.3 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. While the Calyx-based CIRCT flow could initially support simple kernels such as matrix multiplication, it lacked the expressiveness to support more complex ML components like multi-layer perceptrons (MLPs), nonlinear activations, or softmax operations commonly used in attention mechanisms. 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.

3 Technical Contribution

3.1 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 project [7] to compile PyTorch models into MLIR programs. These MLIR programs are then progressively lowered via native MLIR passes to dialects supported by CIRCT. During this stage, we also apply high-level optimizations to improve performance and hardware compatibility.

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.

3.2 Basic Compiler Work in CIRCT and 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—and their hardware realizations.

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 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.

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. Operations on floating-point values are supported by wrapping external libraries. Currently, we use the Berkeley HardFloat library for this purpose. The compiler emits hardware modules corresponding to floating-point operations, wires them into the Calyx program, and correctly maps the semantics of MLIR floating-point ops onto these external modules.

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.

3.3 Optimization Work in CIRCT

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, then, is to expose and maximize parallelism - particularly 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.

Statically optimizing 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: 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` blocks do not perform constant folding, which means that even conditional operations guarded by mutually exclusive conditions will be instantiated in parallel. For example:

Listing 1: Inefficient parallelism in Calyx

```
    par {
        seq {
            if flag_is_true {
                foo;
            }
        }
        seq {
            if flag_is_false {
                bar;
            }
        }
    }
```

Both branches here are compiled to hardware and may contend for resources despite being logically exclusive. This can lead to unintended memory write conflicts.

To illustrate the problem concretely, consider the following simple loop writing to a four-element memory:

Listing 2: Original loop

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

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:

Listing 3: Naive memory banking

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

To parallelize this loop, we materialize it with a factor of 2 using nested `seq` and `par`:

Listing 4: Materialized loop with 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;
        }
    }
}
```

However, each `par` block must be unrolled into separate arms with statically known indices. This produces:

Listing 5: Unrolled parallel execution

```
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;
            }
        }
    }
}
```

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 solve this by restructuring memory accesses. Instead of emitting branches per access, we raise the memory's dimensionality and bake the bank index into the first dimension. For instance:

Listing 6: Bank dimension encoding

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

Unrolling this loop yields:

Listing 7: Conflict-free parallel write

```
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;
        }
    }
}
```

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.

Finally, we observe that loop transformations in hardware are not always semantically equivalent to those in software. Consider:

Listing 8: Sequential outer loop

```
seq for (int i = 0; i < 2; ++i) {
    par for (int j = 0; j < 2; ++j) {
        foo(i, j);
    }
}
```

Listing 9: Parallel outer loop

```
par for (int j = 0; j < 2; ++j) {
    seq for (int i = 0; i < 2; ++i) {
        foo(i, j);
    }
}
```

While equivalent in software, the second version results in hardware duplication. Each parallel arm receives its own FSM for the sequential loop, inflating resource usage. To avoid this, we implement a compiler pass that identifies such patterns and rewrites them to a more efficient schedule.

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

4 Results and Evaluation

4.1 Comparison Across 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.

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 1 shows the wall-clock latency of each model across the two toolchains.

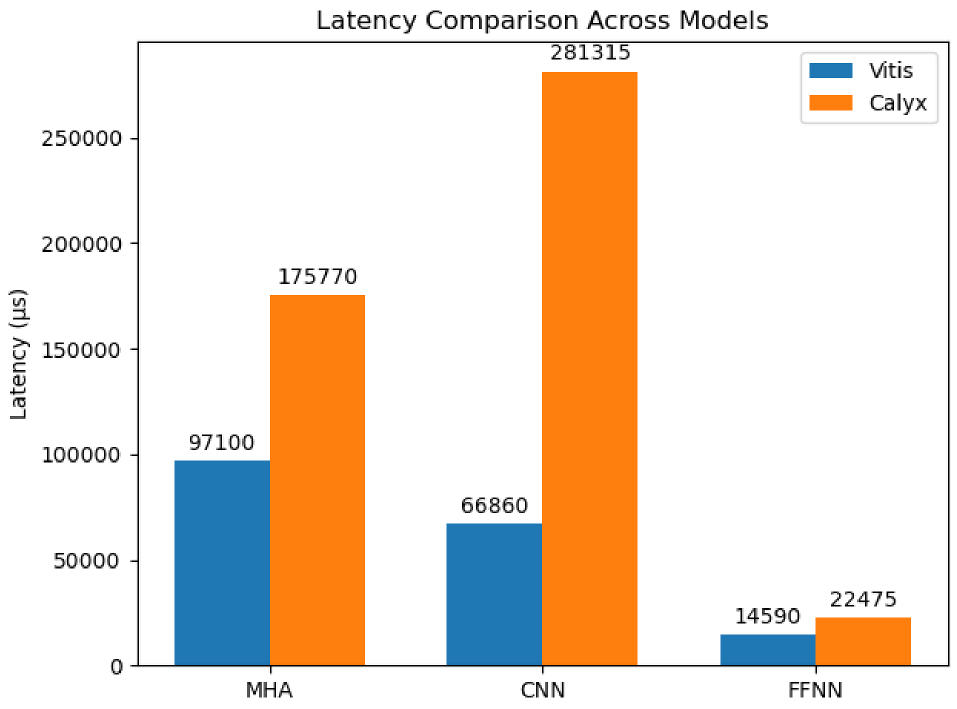

Figure 1: Wall-clock latency comparison across models.

Vitis outperforms our Calyx-based flow in all cases, particularly on the CNN model. 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.

Figure 2 reports resource utilization across the same models.

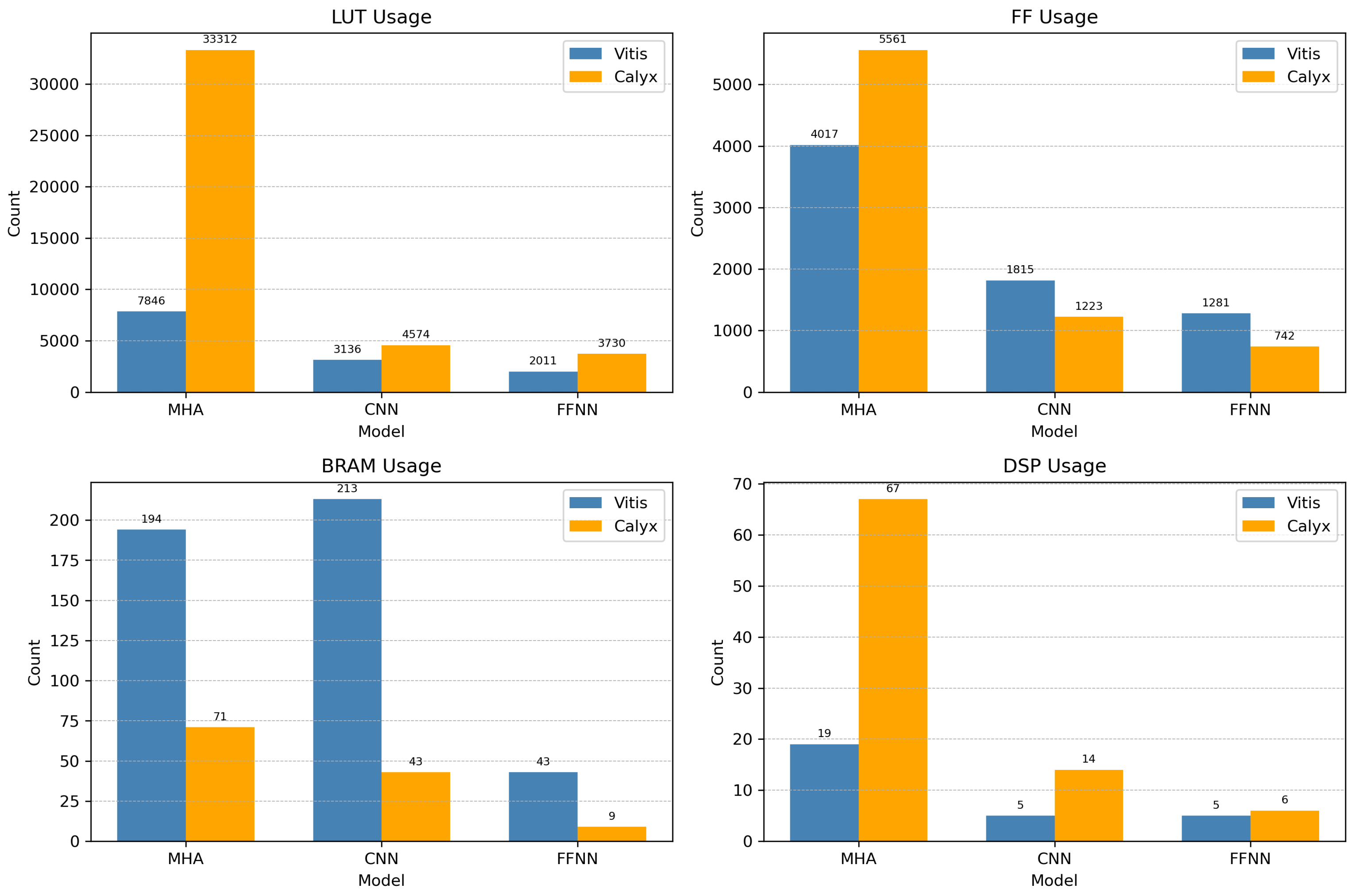

Figure 2: Resource usage comparison across models.

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. Calyx also uses slightly more DSPs. Overall, Calyx incurs higher resource usage—particularly LUTs—because of its verbose control modeling and lack of aggressive scheduling.

5 Case Study: 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.

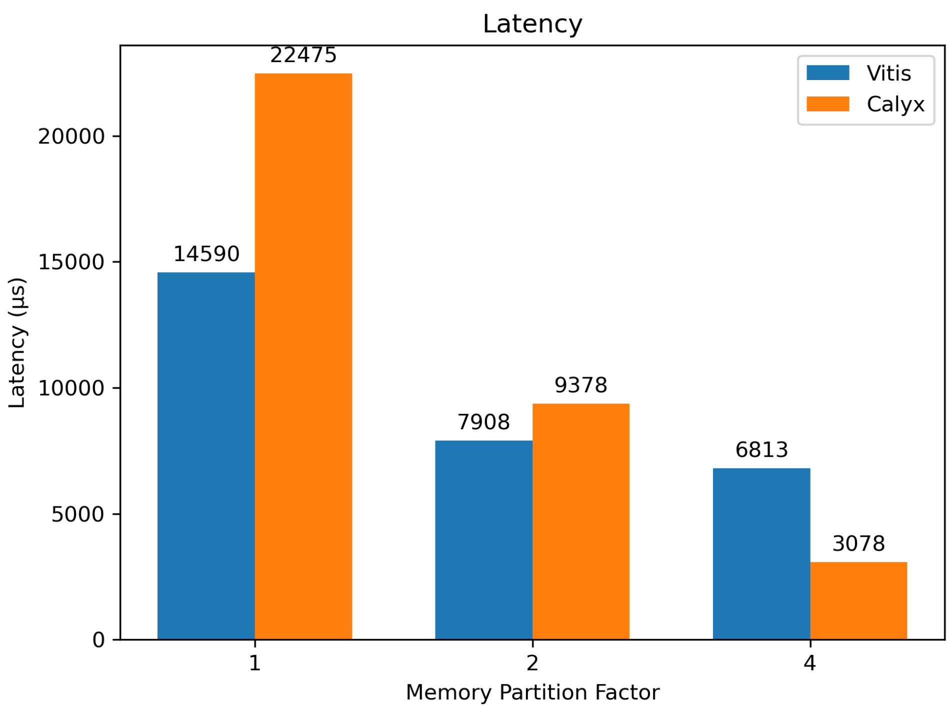

Figure 3: Latency vs. partition factor for FFNN.

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.

Figure 4 shows 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.

6 Conclusion

This work presents a complete open-source compilation toolchain that transforms PyTorch programs into synthesizable hardware designs using MLIR, 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] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47. ACM, 2010.

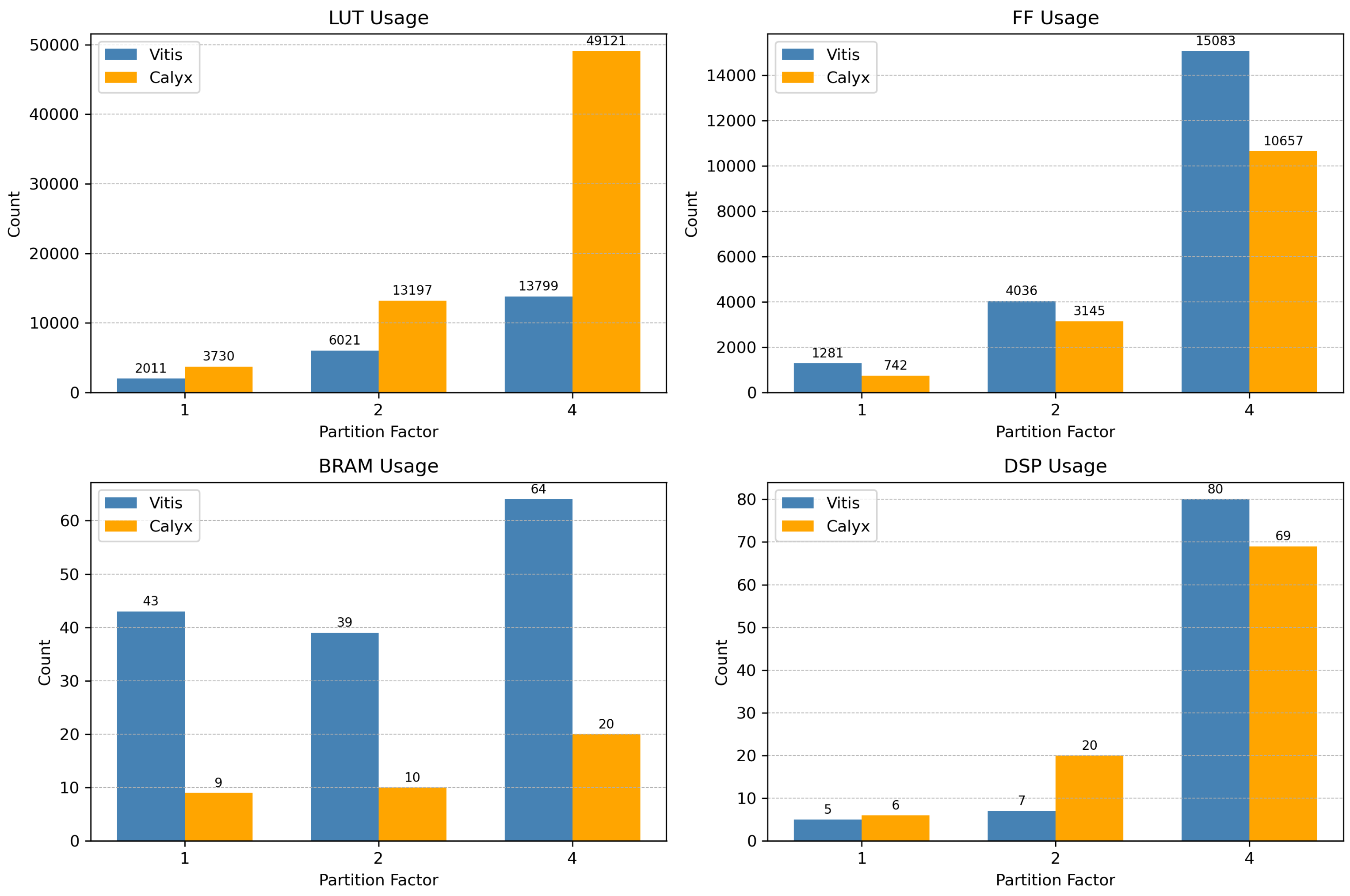

Figure 4: Resource usage vs. partition factor for FFNN.

[2] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap, 2022.

[3] Danish Ali, Anees Ur Rehman, and Farhan Hassan Khan. Hardware accelerators and accelerators for machine learning. In *2022 International Conference on IT and Industrial Technologies (ICIT)*, pages 01–07, 2022.

[4] Muthukumaran Vaithianathan, Mahesh Patil, Shunyee Ng, and Shiv Udkar. Comparative study of fpga and gpu for high-performance computing and ai. 1:37–46, 05 2023.

[5] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 804 – 817, New York, NY, USA, 2021. Association for Computing Machinery.

[6] LLVM CIRCT. "CIRCT" / Circuit IR Compilers and Tools.

[7] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. Allo: A programming model for composable accelerator design. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.