A Functional Approach to Synthesizing Routable Programmable Accelerators for Neural Networks

Tzung-Han Juang
tzung-han.juang@mail.mcgill.ca
McGill University
Montreal, Quebec, Canada

Paul Teng
yuan-po.teng@mail.mcgill.ca
McGill University
Montreal, Quebec, Canada

Christophe Dubach
christophe.dubach@mcgill.ca
McGill University / MILA
Montreal, Quebec, Canada

Abstract

Producing optimized accelerators is tedious, as even modern HDLs (Hardware Description Languages) such as Chisel, require reasoning about low-level concepts. Recent functional approaches, such as Aetherling and SHIR, treat hardware as composition of pure operators. This raises the abstraction level, allowing for systematic optimizations through rewrite rules for FPGAs (Field Programmable Gate Arrays).

These approaches have so far been limited to small, fixed-function accelerators. Recent work maps neural networks to FPGAs by sharing coarse-grained functions via the *Let* construct. However, as the number of call sites or parallelism increases, synthesis fails due to increased routing congestion.

These limitations are addressed with a new way to express sharing in a functional IR (Intermediate Representation). By combining the *Reduce* and *SwitchApply* primitives over an instruction stream, functions become programmable, with shared control logic and a datapath, reducing routing pressure. Upper-bounded streams further enable sharing across varying input sizes. Across networks from LeNet 5 to ResNet, the resulting FPGA designs remain routable, delivering high performance with speedups between 1.1×–3.4× compared to prior work.

CCS Concepts: • **Hardware → Hardware accelerators**; • **Software and its engineering → Functional languages**; *Source code generation.*

Keywords: High-Level Synthesis, Functional Programming, Neural Networks

ACM Reference Format:
Tzung-Han Juang, Paul Teng, and Christophe Dubach. 2026. A Functional Approach to Synthesizing Routable Programmable Accelerators for Neural Networks. In *Proceedings of the 27th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '26), June 15–16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3814943.3816183

LCTES '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2721-4/2026/06
https://doi.org/10.1145/3814943.3816183

1 Introduction

Designing hardware accelerators is a complex and manual process. Designers typically use low-level HDLs such as VHDL or Verilog, managing signals, memory, and control explicitly. Higher-level frameworks such as Chisel [3], Lava [4], μFP [28], and Cλash [2] provide modern abstractions, but low-level hardware details persist. These limitations highlight the need for hardware generation with higher-level abstractions. C/C++ and OpenCL HLS (High-Level Synthesis) offer a good starting point but suffer from unpredictable performance compared to functional approaches [20].

Recent approaches, such as Spatial [16], Lift-HLS [17], Aetherling [7], and SHIR [26], advocate functional high-level abstractions to simplify accelerator design. While Spatial supports a limited set of parallel patterns, Lift-HLS, Aetherling, and SHIR rely on composing operators like *Map* and *Reduce*, which act on stream- and vector-based data types, naturally exposing pipeline and spatial parallelism. A key advantage is that optimizations can be expressed as rewrite rules, enabling a systematic, principled approach to performance tuning.

So far, these composable approaches have been limited to small, *fixed-function* accelerators. Recent work [14] maps entire neural networks to FPGAs by sharing functions such as dot products via the *Let* construct. However, this does not scale: as call sites, parallelism, and operand bit width increase, routing congestion quickly dominates as control logic is scattered across call sites, making designs difficult or impossible to synthesize. Furthermore, they offer little flexibility and programmability: an accelerator designed for ResNet 18 will be unable to execute ResNet 50 due to the difference in layer configurations and data sizes.

This work addresses these limitations by *expressing* programmable accelerators in a compositional functional IR for HLS — while leveraging existing approaches. Most existing functional IR building blocks are reused with only minor extensions: 1) A classical *Reduce* is repurposed to iterate over an instruction stream, repeatedly applying a function to provide programmability and enable sharing; 2) A new upper-bounded stream is introduced to handle variable-length data; and 3) A simple *SwitchApply* construct is presented, allowing multiple functions to share the same datapath. Expressing accelerators in this way enables systematic HLS optimizations via rewrite rules. This produces programmable designs

$$\mathit{TupleN} : T_1 \mapsto \cdots \mapsto T_N \mapsto T_1 \to \cdots \to T_N \to (T_1, \ldots, T_N)$$

$$\mathit{Get} : T_1 \mapsto \cdots \mapsto T_N \mapsto I^{Nat \le N} \mapsto (T_1, \ldots, T_N) \to T_I$$

$$\mathit{Counter} : N \mapsto {}^{\text{STM}}[\mathit{Int}_{\lceil \log_2 N \rceil}]_N$$

$$\mathit{VecGen} : T \mapsto N \mapsto T \to {}^{\text{VEC}}[T]_N$$

$$\mathit{MapStm} : T \mapsto U \mapsto N \mapsto (T \to U) \to {}^{\text{STM}}[T]_N \to {}^{\text{STM}}[U]_N$$

$$\mathit{MapVec} : T \mapsto U \mapsto N \mapsto (T \to U) \to {}^{\text{VEC}}[T]_N \to {}^{\text{VEC}}[U]_N$$

$$\mathit{ReduceStm} : T \mapsto U \mapsto N \mapsto ((U, T) \to U) \to U \to {}^{\text{STM}}[T]_N \to U$$

$$\mathit{ZipStm} : T \mapsto U \mapsto N \mapsto {}^{\text{STM}}[T]_N \to {}^{\text{STM}}[U]_N \to {}^{\text{STM}}[(T, U)]_N$$

$$\mathit{ZipVec} : T \mapsto U \mapsto N \mapsto {}^{\text{VEC}}[T]_N \to {}^{\text{STM}}[U]_N \to {}^{\text{VEC}}[(T, U)]_N$$

$$\mathit{SplitStm} : T \mapsto N \mapsto M \mapsto {}^{\text{STM}}[T]_N \to {}^{\text{STM}}[{}^{\text{STM}}[T]_M]_{N/M}$$

$$\mathit{SplitVec} : T \mapsto N \mapsto M \mapsto {}^{\text{VEC}}[T]_N \to {}^{\text{VEC}}[{}^{\text{VEC}}[T]_M]_{N/M}$$

$$\mathit{JoinStm} : T \mapsto N \mapsto M \mapsto {}^{\text{STM}}[{}^{\text{STM}}[T]_N]_M \to {}^{\text{STM}}[T]_{N*M}$$

$$\mathit{JoinVec} : T \mapsto N \mapsto M \mapsto {}^{\text{VEC}}[{}^{\text{VEC}}[T]_N]_M \to {}^{\text{VEC}}[T]_{N*M}$$

$$\mathit{Repeat} : T \mapsto N \mapsto T \to {}^{\text{STM}}[T]_N$$

$$\mathit{VecToStm} : T \mapsto N \mapsto {}^{\text{VEC}}[T]_N \to {}^{\text{STM}}[T]_N$$

$$\mathit{StmToVec} : T \mapsto N \mapsto {}^{\text{STM}}[T]_N \to {}^{\text{VEC}}[T]_N$$

Figure 1. Most common primitives in SHIR. $T \to U$ is a function type. $T \mapsto U$ represents a type function, which given a type parameter T returns type U. $(T_1, \ldots, T_N)$ is a N-ary tuple type. ${}^{\text{STM}}[T]_N$ and ${}^{\text{VEC}}[T]_N$ represent a stream and a vector, respectively.

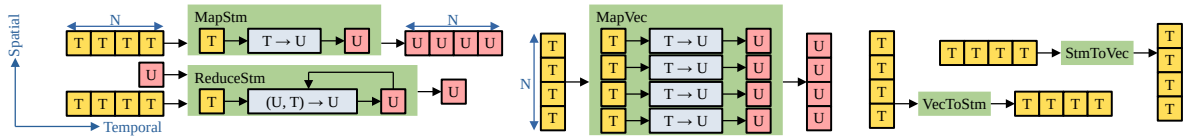


Figure 2. Dataflow of common SHIR functional primitives.

that are easier to route and more flexible while preserving a high-level, principled functional programming model.

Once the accelerator has been designed using the functional approach, a PyTorch runtime and backend are provided. Given a neural network and the accelerator's instruction format, it emits the instruction sequence and handles end-to-end execution. This approach is evaluated on four network classes, LeNet 5, VGG, Tiny YOLOv2, and ResNet, synthesizing all accelerators on a real FPGA platform. The results show these accelerators achieve competitive performance while avoiding routing issues that limited prior work.

To summarize, the contributions of this paper are:

- Identify limitations of *Let*-based resource sharing;
- Proposes a compositional approach to programmable hardware using reduction and switching constructs;
- Extends the notion of streams with upper-bounded streams to support variable-length data;
- Integrates with the popular PyTorch workflow;
- Evaluates the flow on a real FPGA, demonstrating efficient and routable accelerators for neural networks.

2 Background

Functional approaches can naturally express fixed-function accelerators. A review is provided of SHIR [14, 26, 27], a state-of-the-art functional HLS framework on which this work builds.

2.1 Functional IR in SHIR

SHIR is based on System $F_{<:}$, enabling bounded type parameters. Like Aetherling, SHIR [26] expresses hardware as compositions of functional primitives operating on streams or vectors. Stream-based operations process elements sequentially, while vector-based operations handle multiple elements in parallel. SHIR is more general than Aetherling and also supports memory operations. To improve performance, SHIR automatically applies rewrite rules [26, 27], including fusing operations, converting streams to vectors to increase parallelism, and eliminating costly data transformations.

2.1.1 Tuples. SHIR supports tuples of arbitrary length via the *TupleN* primitive (figure 1). The compiler takes a Scala sequence of types as input and emits the corresponding N-ary tuple type. Elements can be extracted with *Get*, where the position is statically known (type parameter I: a $Nat \le N$).

2.1.2 Stream and Vector Operations. SHIR supports operations on streams, processed in time, and vectors, processed in space (figure 1). Stream operations are dynamically synchronized in hardware via a handshake protocol to ensure the consumer is ready when valid data is produced. A last signal indicates the end of a stream.

The *Counter* and *VecGen* primitives initialize a stream or vector of N elements of type T. Vectors carry their length in the type, which is required to instantiate the correct number of functional units in a *MapVec*. Similarly, streams carry their length in the type, for example, when converting between vectors and streams. Later, this paper shows how to lift this restriction for streams to enable more flexible designs.

Operations include standard primitives (map, reduce, zip), data reshaping (split, join, repeat), and stream–vector conversions. A visual overview is shown in figure 2.

2.1.3 Memory Operations. SHIR introduced memory operations explicitly in the IR. While both local on-chip RAM

$$Alloc : N \mapsto T \mapsto \overset{\text{RAM}}{}[T]_N$$
$$Read : N \mapsto T \mapsto \overset{\text{RAM}}{}[T]_N \rightarrow Nat^{\leq N} \rightarrow T$$
$$Write : N \mapsto T \mapsto \overset{\text{RAM}}{}[T]_N \rightarrow Nat^{\leq N} \rightarrow T \rightarrow \overset{\text{RAM}}{}[T]_N$$

(a) High-level operations.

$$Read : Int \rightarrow Int \rightarrow \overset{\text{VEC}}{}[Byte]_{64}$$
$$Write : Int \rightarrow Int \rightarrow \overset{\text{VEC}}{}[Byte]_{64} \rightarrow Int$$

(b) Low-level operations.

Figure 3. Host memory primitives in SHIR [26].

```
mem1 = Alloc<1024,Int8>;
mem2 = Alloc<1024,Int8>;
mem3 = Alloc<1024,Int8>;
ReduceStm(
 λ (acc, (sum,idx)) =>
   Write(acc,idx,sum),
 mem3,
 ZipStm(
  MapStm(
   +,

   ZipStm(
    MapStm(
     Read(mem1,_),
     Counter<1024>),
    MapStm(
     Read(mem2,_),
     Counter<1024>))
  ),
  Counter<1024>))
```

(a) High-level expression.

```
baseAddr1 = 0    /64;
baseAddr2 = 1024/64;
baseAddr3 = 2048/64;
ReduceStm(
 λ (acc,(sum,idx)) =>
   Write(acc,idx,sum),
 baseAddr3,
 ZipStm(
  MapStm(λ(v1,v2) =>
   MapVec(+,
     ZipVec(v1,v2)),
   ZipStm(
    MapStm(
     Read(baseAddr1,_),
     Counter<1024/64>)
    MapStm(
     Read(baseAddr2,_),
     Counter<1024/64>))))
  ),
  Counter<1024/64>)
```

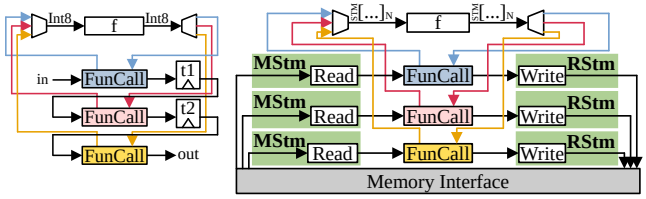
(b) Low-level expression.

Figure 4. SHIR expressions for adding two arrays in memory. `x = a; e` is syntactic sugar for a Let construct: `Let x = a in e`. Type parameters are passed inside <> while function arguments are passed inside `()`.

and global off-chip host RAM are supported by SHIR, only host RAM operations are reviewed here due to limited space. SHIR uses a multi-level IR, where memory operations are lowered step by step, closer to the reality of hardware.

High-Level. At a high level, SHIR provides a memory type and three main operations (figure 3a). *Alloc* creates a memory for N elements of type T. *Read* takes a memory and a positive index $\leq N$ and returns the data, while *Write* returns an updated memory given a memory, index, and data. The programmer must ensure that a memory variable is not reused after a *Write*; this could be enforced with a linear type system, but that is beyond this paper's scope.

Figure 4a shows an example using these primitives. It computes the sum of two arrays of 1024 Int8 values and stores the result in memory. Each array is read by mapping over a counter to generate indices, and the resulting streams are zipped for addition. The output is written back using a reduction over writes, where the accumulator holds the new memory state. This pattern, introduced in prior work [26], ensures that the input memory is never reused after a write.



(a) Registers only. **(b)** Data flows through memory.

```
f : Int8 → Int8  = ...;  // shared function
t1=f(in); t2=f(t1); out=f(t2);  // call chain
```

(c) Functional expression corresponding to (a).

```
f : MS[...]N → MS[...]N  = ...;  // shared function
baseIn=0; baseT1=N; baseT2=2N; baseOut=3N; // base
     addr

T1 = ReduceStm( // T1 = f(In);
  λ(acc,(data,idx)) => Write(acc,idx,data),
  baseT1,                        // initial accumulator
  ZipStm(f(MapStm(Read(baseIn,_),
    Counter<N>)), Counter<N>)); // rd and wr indices

T2 = ReduceStm( // T2 = f(T1);
  λ(acc,(data,idx)) => Write(acc,idx,data),
  baseT2,                        // initial accumulator
  ZipStm(f(MapStm(Read(T1,_),
    Counter<N>)), Counter<N>)); // rd and wr indices

Out = ReduceStm( // Out = f(T2);
  λ(acc,(data,idx)) => Write(acc,idx,data),
  baseOut,                       // initial accumulator
  ZipStm(f(MapStm(Read(T2,_),
    Counter<N>)), Counter<N>));  // rd and wr indices
```

(d) Functional expression corresponding to (b).

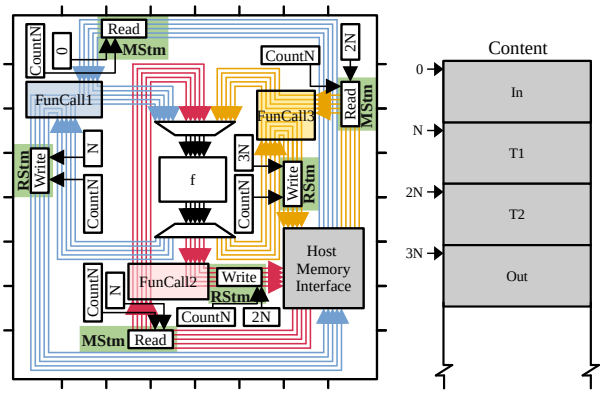
Figure 5. How functions are shared in SHIR.

Low-Level. At the low level, SHIR exposes two operations for interfacing with host RAM (figure 3b). *Read* takes a base address and an index, returning a 64-byte vector matching the PCI-Express bus granularity. *Write* takes a base address, index, and 64 bytes of data, and returns the base address to facilitate correct synchronization.

The SHIR compiler lowers the high-level IR to the low-level one using rewrite rules [26]. This involves lowering each memory operation and coarsening them to a 64-byte granularity. Padding is inserted automatically if needed.

Figure 4b shows the lowered code to the high-level code in figure 4a. Allocations are replaced with explicit base addresses, and *Read* and *Write* now use these addresses. The simple addition of two streams is replaced with an expression that adds streams of 64-element Int8 vectors. Finally, the counters produce 64 times fewer values, since each read or write now operates on 64 elements at once.

As explained, the low-level *Write* returns the base address, simplifying lowering by allowing each memory object to be replaced with its address. It is also used by reductions to synchronize *Write* execution via the handshake protocol, forcing each write to complete before the next accumulator update and ensuring sequential consistency.



(a) Chip view with routing congestion. **(b)** Memory view.

Figure 6. Sharing a function across three call sites and accessing data from the host memory.

2.2 Sharing Function with Let

Sharing and reusing functional units is crucial for efficient hardware accelerators, especially for workloads like neural networks where many layers perform similar operations. Prior work [14] achieves this functionally using a *Let* construct. With data, the SHIR compiler introduces a register; with a function, it instantiates a single functional unit and adds multiplexers, demultiplexers, and arbitration logic to route data and ensure only one call occurs at a time.

Sharing Scalar Functions. Figure 5a shows a shared function called three times on a single Int8 value, with the corresponding expression in figure 5c. Data is exchanged between successive calls via registers.

Sharing Streaming Functions. When a function produces a stream, the data must be stored in memory, and the function must fully output the stream, over multiple cycles, before it can process the next one. Figure 5b shows a simplified dataflow for a function shared three times, and the corresponding expression is in figure 5d. Three reductions write the two temporary results and the final output to memory, with each reduction feeding data into the function from the input or previously computed results.

3 Enabling Programmable Hardware

Although function sharing can be expressed in a functional IR, it can lead to severe routing issues during hardware synthesis, as we will see next.

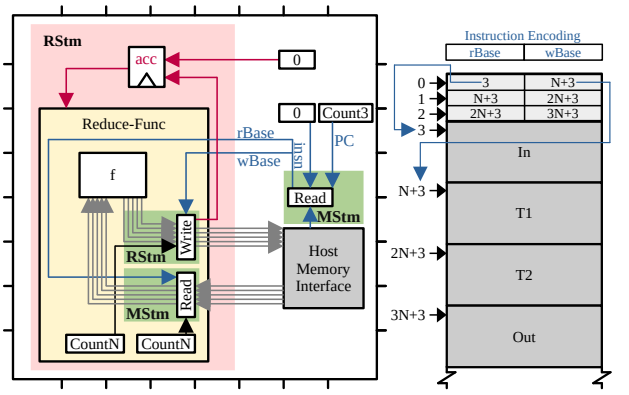
3.1 Limitations of Let-Sharing

Prior work [14] applied Let-sharing to small networks like VGG and Tiny YOLO. However, larger applications can cause routing congestion as function calls increase.

Figure 6 shows a possible physical layout and memory contents for the shared-function example from figure 5d. All components, including the shared function f, its call sites, and memory interfaces, must be placed and routed. As the number of function calls grows, the FPGA synthesizer may fail to route all connections, and the problem worsens with higher parallelism as more data bits are required.

This problem is further exacerbated when different input sizes are used with a shared function. Extra hardware components need to be added to tile and pad the data to make it fit with the fixed-sized shared function, which adds pressure on the FPGA synthesizer. Finally, applications such as neural networks exhibit multiple types of layers (*e.g.,* fully connected, activation, pooling). This offers opportunities to share different functions but leads to further routing issues.

The following sections show how these issues can be addressed with minimal extensions to existing functional primitives, the main contribution of this work.



(a) Chip view. **(b)** Memory view.

```
1 f : [...]_N → [...]_N = ...;
2 PC = Counter<3>; insns = MapStm(Read(0,_), PC);
3 ReduceStm(λ (acc,insn) =>
4   ReduceStm(λ(ba,(x,i)) => Write(ba,i,x),
5     insn.wBase,
6     ZipStm(
7       f(MapStm(Read(insn.rBase,_), Counter<N>)),
8       Counter<N>)),
9   0, insns)
```

(c) Expression for Reduce-sharing.

Figure 7. Using Reduce to avoid multiplexers in Let-sharing.

3.2 Sharing with Reduce

Routing issues arise because many call sites connect to the same hardware component. To address this, a new function-sharing approach is introduced as an alternative to *Let*. The first step is to consider repeated calls to a single function on fixed-size data, then generalize to variable input sizes and multiple functions.

The key idea is to use *Reduce* to express repeated function applications. As discussed in section 2.2, shared functions generally require data to pass through memory. A reduction is therefore performed over a stream of base addresses for

$$UCounter : N \mapsto Nat^{\leq N} \to \overset{\text{STM}}{\lceil Int_{\lceil \log_2 N \rceil} \rceil_N}$$
$$MapUStm : T \mapsto U \mapsto N \mapsto (T \to U) \to \overset{\text{STM}}{\lceil T \rceil_N} \to \overset{\text{STM}}{\lceil U \rceil_N}$$
$$ReduceUStm : T \mapsto U \mapsto N \mapsto ((U,T) \to U) \to U \to \overset{\text{STM}}{\lceil T \rceil_N} \to U$$
$$ZipUStm : T \mapsto U \mapsto N \mapsto \overset{\text{STM}}{\lceil T \rceil_N} \to \overset{\text{STM}}{\lceil U \rceil_N} \to \overset{\text{STM}}{\lceil (T,U) \rceil_N}$$
$$SplitUStm : T \mapsto N \mapsto M \mapsto \overset{\text{STM}}{\lceil T \rceil_N} \to \overset{\text{STM}}{\lceil} \overset{\text{STM}}{\lceil T \rceil_M} \rceil_{N/M}$$

$$SplitUStm : T \mapsto N \mapsto M \mapsto Nat^{\leq M} \to \overset{\text{STM}}{\lceil T \rceil_N} \to \overset{\text{STM}}{\lceil} \overset{\text{STM}}{\lceil T \rceil_M} \rceil_{N/M}$$
$$JoinUStm : T \mapsto N \mapsto M \mapsto \overset{\text{STM}}{\lceil} \overset{\text{STM}}{\lceil T \rceil_N} \rceil_M \to \overset{\text{STM}}{\lceil T \rceil_{N*M}}$$
$$UStmToVec : T \mapsto N \mapsto \overset{\text{STM}}{\lceil T \rceil_N} \to \overset{\text{VEC}}{[(T, Bool)]_N}$$
$$PackVec : T \mapsto N \mapsto \overset{\text{VEC}}{[(T, Bool)]_N} \to \overset{\text{VEC}}{[(T, Bool)]_N}$$
$$VecToUStm : T \mapsto N \mapsto \overset{\text{VEC}}{[(T, Bool)]_N} \to \overset{\text{STM}}{\lceil T \rceil_N}$$
$$RepeatUStm : T \mapsto N \mapsto Nat^{\leq N} \to T \to \overset{\text{STM}}{\lceil T \rceil_N}$$

Figure 8. New primitives for upper-bounded streams. $\overset{\text{STM}}{\lceil T \rceil_N}$ represents (ceiling brackets) a dynamically sized stream with a maximum length of N. $Nat^{\leq N}$ is a scalar type restricted to values less than or equal to N.

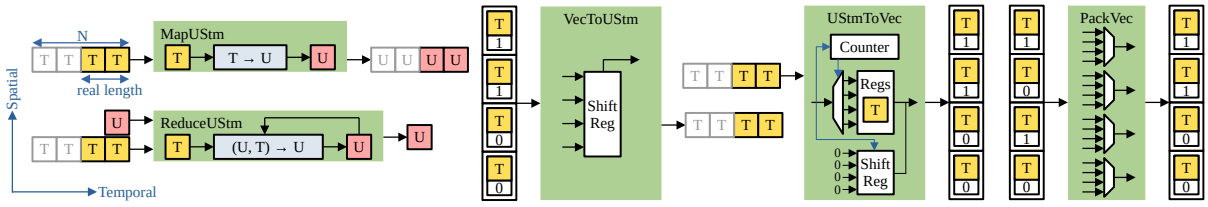


Figure 9. Dataflow and hardware details for some of the upper-bounded stream primitives. Cells with a gray border/content do not contribute to the processing time of streams.

inputs and intermediate results. Each iteration reads data from memory, applies the shared function, and writes back the result. This simple approach achieves three goals:

1. it ensures the shared function is only used once at a time, removing the need for any arbitration;
2. it reuses the same hardware logic to read and write data from/to memory;
3. it removes the need for multiplexing data.

The previous example (figure 5d) is re-expressed using a reduction (figure 7c). A PC (program counter) generates three addresses (0, 1, 2). In memory, these locations store the addresses for reading and writing the data processed by the shared function (figure 7b). These act as *instructions*, making the hardware programmable.

With the instruction stream providing the relevant memory locations, the reduction is applied on them. The accumulator is initialized to 0, but its value is irrelevant as it merely acts as a synchronization point to ensure each iteration completes before the next begins. At each step, data is read from memory at `rBase`, processed by `f`, and the result is written back starting at `wBase`.

The resulting hardware is shown in figure 7a. It performs the same functionality as the earlier design using *Let*. Critically, routing issues disappear because a single hardware unit executes f and handles all memory reads and writes. This design is also more flexible, as memory base addresses can now be *programmed* in software.

A remaining limitation is that f must process the same amount of data, as its type fixes the stream length to N. The number of instructions, such as 3 in our example, is also hard-coded in hardware. The next section shows how these restrictions are lifted with a new type of stream.

3.3 Upper-Bounded Stream

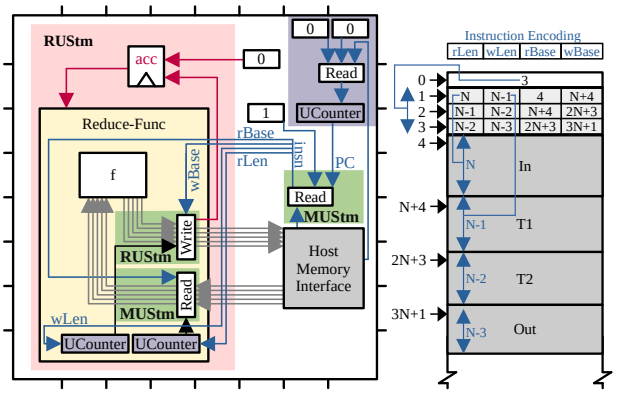
Streams traditionally model sequential, element-by-element data. However, prior systems such as SHIR [26] and Aetherling [7] require stream lengths to be fixed at compile time, preventing shared functions operating on streams from supporting multiple input sizes. To overcome this limitation, an upper-bounded stream type is now introduced, allowing different stream lengths to be represented at run time.

3.3.1 Type. The upper-bounded stream type $\overset{\text{STM}}{\lceil T \rceil_N}$ specifies a stream of *at most* N elements of type T. Because a fixed stream $\overset{\text{STM}}{[T]_N}$ is simply the special case where the run-time length equals the bound, it is redefined as a subtype of the upper-bounded stream.

3.3.2 Primitives. Figure 8 introduces the primitives for the upper-bounded stream type. Most closely mirror their fixed-stream counterparts. For instance, *MapUStm* applies a function to each element of the stream, just like *MapStm*, as illustrated in figure 9. The difference is that the processing time now depends on the run-time length of the stream. This is enabled in hardware by the streaming handshake protocol, which uses a `last` signal to mark the last element.

UCounter is the upper-bounded counterpart to *Counter* (figure 1): it requires a run-time length. Similarly, *RepeatUStm* and one variant of *SplitUStm* accept this extra input. There are two *SplitUStm*s: fixed-chunk and upper-bounded-chunk. If users can guarantee the chunk size is constant, the fixed version reduces resource overhead.

3.3.3 Upper-Bounded Stream and Vector Conversion. A key difference from fixed streams appears in vector/stream conversions and the *PackVec* primitive. Since vectors have

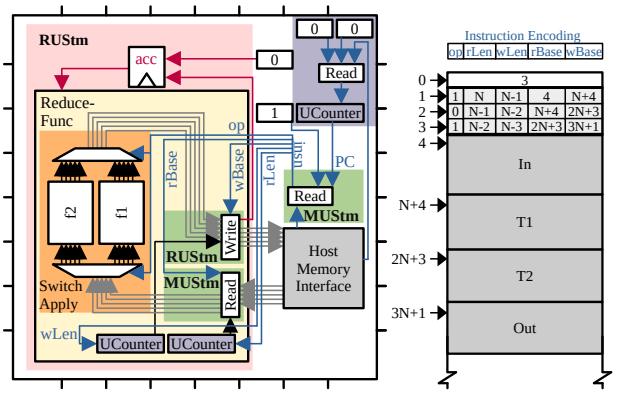


(a) Chip view. **(b)** Memory view.

```
f : ⟦...⟧_N → ⟦...⟧_{N-1} = ...;
insnNum = Read(0,0);
PC = UCounter(insnNum); insns = MapUStm(Read(1,_),PC);
ReduceUStm(λ (acc,insn) =>
  ReduceUStm(λ(ba,(x,i)) => Write(ba,i,x),
    insn.wBase,
    ZipUStm(
      f(
        MapUStm(Read(insn.rBase,_), UCounter(insn.rLn)
      )
        ),
      UCounter(insn.wLn))),
  0, insns)
```

(c) Expression with new primitives.

Figure 10. Sharing a function with upper-bounded streams.



(a) Chip view. **(b)** Memory view.

```
f1: ⟦...⟧_N → ⟦...⟧_{N-1} = ...;
f2: ⟦...⟧_N → ⟦...⟧_{N-1} = ...;
insnNum = Read(0,0);
PC = UCounter(insnNum); insns = MapStm(Read(1,_),PC);
ReduceUStm(λ (acc,insn) =>
  ReduceUStm(λ (ba,(x,i)) => Write(ba,i,x),
    insn.wBase,
    ZipUStm(
     SwitchApply(f1, f2,
     MapUStm(Read(insn.rBase,_), UCounter(insn.rLen)),
       insn.op),
     UCounter(insn.wLen)),
  0, insns)
```

(c) Expression with two functions.

Figure 11. Sharing multiple functions.

fixed length, converting an upper-bounded stream produces a vector of the stream's upper bound, pairing each element with a flag. The flag is true for the first n elements, where n is the stream's run-time length, and false for the rest.

When converting such a vector back to an upper-bounded stream, only the first n elements whose associated boolean value is true will be produced in the stream. Everything from the first false value will be ignored. This approach allows for an efficient hardware design requiring a single shift register for this primitive.

When vector elements need reordering, the *PackVec* primitive packs elements with a true flag to the front while preserving their relative order (figure 9). Packing is costly in hardware due to the many multiplexers. By separating conversion and packing, users can skip packing if the vector is already guaranteed packed, saving significant resources.

3.3.4 Reduce-Sharing with Upper-Bounded Stream.

With the new upper-bounded streams and primitives, we can express a more programmable accelerator. As can be seen in figure 10c, the data lengths for memory reads and writes can now come from memory rather than being fixed, and the number of instructions can also be read from memory (location 0). This is enabled by the *UCounter* primitive, which produces an upper-bounded stream from a run-time value, though a static upper bound is still needed to size the output.

The rest of the expression is unchanged, except for using upper-bounded stream operations. On the hardware side (figure 10a), the read (`rLn`) and write (`wLn`) lengths now come from the memory interface.

3.4 Sharing with Multiple Functions

The previous sections showed how a single function can be shared and applied to data of varying lengths. The final step is supporting multiple functions, enabled by the new *SwitchApply* primitive, which selects and applies a function from a pool based on a run-time index.

$$SwitchApply : T \mapsto U \mapsto \underbrace{(T \to U) \to \cdots \to (T \to U)}_{N \text{ times}} \to T \to Nat^{<N} \to U$$

It takes a pool of N functions, a run-time value of type T, and a zero-based run-time index specifying which function to apply. This signature is generated using the same mechanism as the *TupleN* primitive.

In terms of the hardware implementation, the forwarded argument of type T is always connected to all functions in the switching pool, but only the function selected by the run-time index receives the actual handshake signals from the surrounding context: the valid signal from the argument producer and the ready signal from the *SwitchApply* consumer. For all other functions, both the valid and ready signals are

explicitly set to false to block the execution. Additional multiplexers are used to redirect the relevant output and other handshake signals from the selected function. The number of multiplexers instantiated in the current implementation scales linearly in terms of the number of functions in the switching pool.

Note that an alternative approach could be to use a standalone switch primitive composed with function calls. However, this would generate more complex hardware, since each call would need separate control logic.

The running example can now support multiple functions. Figure 11c shows the updated form, using *SwitchApply* to select between f_1 and f_2, with hardware and memory layouts in figures 11a and 11b. The outer lambda λ $(acc, insn)$ can grow arbitrarily complex, for example, by chaining functions in a pipeline or nesting multiple *SwitchApply*s.

3.5 Optimizations

The SHIR compiler already performs several rewrite-based optimizations [27]. Most existing rewrites apply to upper-bounded streams with minimal changes. For example, map fusion $MapUStm(f) \circ MapUStm(g) \implies MapUStm(f \circ g)$ remain valid with upper-bounded streams. Split-join fusion $SplitUStm \circ JoinUStm \implies \varnothing$ also remains valid as the constant or dynamic split factor supplied to the *SplitUStm* primitive is expected to perfectly divide the length of the original input stream. The new types and primitives also open opportunities for further optimizations.

3.5.1 Upper-Bounded Stream Elimination. While upper-bounded streams behave like fixed streams, their primitives are costlier on the FPGA due to extra handshaking complexity. Hence, replacing upper-bounded streams with fixed streams whenever possible is desirable.

Since all upper-bounded streams stem from *UCounter*s, a rewrite can detect when a *UCounter* has a compile-time value and replace it with a fixed counter. Then, any upper-bounded stream primitive whose inputs are all fixed streams can be converted to a fixed stream primitive.

For instance, the following expression that uses a *UCounter* with a constant will be turned into a version that no longer uses an upper-bounded stream map primitive:

$$MapUStm(f, UCounter(3)) \implies MapStm(f, Counter\langle 3\rangle)$$

This optimization is useful in combination with others as we will shortly see.

3.5.2 Constant Propagation. Some functional expressions for the programmable hardware use helper macros that introduce more flexibility than needed. Consequently, certain expressions (e.g., instruction fields) come from static constants rather than run-time data. Constant propagation moves these constants closer to their use sites, simplifying the resulting expression.

```
ReduceUStm(
 λ (base, (data, offs)) => Write(base, offs, data),
  insn.wBase,
 ZipUStm(
  SwitchApply(λ x => x, AvgPool),
   Requant(insn.qsBase, insn.qz,
    Bias(insn.bBase, insn.bLen,
    PartialSum(insn.opPsum, ParallelDP(
     SwitchApply(ReadConvImages, ReadFCWeight, insn,
      insn.opCfc),
     SwitchApply(ReadConvWeight, ReadFCImages, insn,
      insn.opCfc))))),
    insn.opPool),
   WriteAddr(insn.wBase, insn.wLen)))
```

Figure 12. An overview of the functional expression for the LeNet 5 programmable hardware.

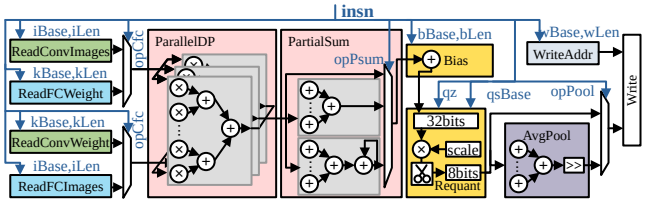


Figure 13. Instruction reduction loop's body for figure 12.

In the following example, the rewrite moves the constant $Constant\langle k\rangle$ inwards so that it can potentially trigger other simplification rewrites.

$$MapUStm(\lambda y \Rightarrow f(y.1, y.2), RepeatUStm(Tuple(a, Constant\langle k\rangle), n)) \implies MapUStm(\lambda z \Rightarrow f(z, Constant\langle k\rangle), RepeatUStm(a, n))$$

This optimization is useful with upper-bounded stream elimination: if a constant reaches a *UCounter*, it triggers further rules to remove upper-bounded stream operations.

4 Expressing Neural Networks

The new proposed approach allows expressing neural networks as programmable hardware. Here, LeNet 5 [18] is used to demonstrate this. It takes a 28x28 grayscale image of a digit, feeds it through two pooled convolutions, then feeds it through three fully connected layers, classifying the original digit in the image.

4.1 Decomposing into Operations

A naive approach could implement each layer as a separate function in a *SwitchApply* within a reduction (figure 11a). It is inefficient since convolution and fully connected layers share scarce DSPs (Digital Signal Processors), limiting parallelism.

Instead, layers are decomposed into smaller operations, whose behavior is controlled by instructions to implement each layer. The resulting instruction reduction loop is shown in figure 12, with the corresponding hardware in figure 13.

Data Generators. The key difference between convolution and fully connected layers is the ordering and repetition

Table 1. A summary of the size of the operations, in terms of IR nodes, and the amount of times each operation is present in the LeNet 5 programmable hardware design.

Operation	Size	Cnt
Instruction Reduction		
Read	26	1
Reduction	1	1
Compute		
Parallel dot product	34	1
Partial sum	26	3
Bias add	10	2
Requant rescale	18	1
Activation ReLU	1	1
Pooling average	42	1
Read Access		
Conv image (shallow)	39	1
Conv weight (shallow)	39	1
FC image (shallow)	49	1
FC weight (shallow)	60	1
Bias read	47	1
Requant read	53	1
Write Access		
Output (shallow)	13	1
Total		18

of the input data. Hence, two pairs of different "data generators" are made to handle the weights and images. Both feed their outputs, through a *SwitchApply* primitive, into `ParallelDP` and `PartialSum` on lines 8–12.

Parallel Dot Products. `ParallelDP` performs multiple dot products in parallel. It takes two 2D streams of 2D vectors, splits them into 1D vectors for the dot products, and outputs a 2D stream of 1D vectors. The dot product dominates DSP usage on the FPGA.

Partial Sum. The `PartialSum` block supports three modes: pass the input vector unchanged, partially sum based on input orientation, or fully sum and accumulate. These modes accommodate both large and small layers, including convolution and fully connected layers.

Other Components. The rest of the pipeline adds the bias, requantizes, and writes the result to memory. These steps are straightforward and are handled by upper-bounded primitives. As pooling always follows convolution, it is integrated directly into the pipeline.

4.2 Implementation Effort

Each operation is implemented using SHIR expressions. Here, the expression size, measured by the number of main IR nodes (excluding core nodes such as *FunCall*s, tuple accesses, and lambdas), is used as a proxy for manual effort. The LeNet 5 operations and their sizes are listed in table 1.

While some operations may seem complex, the effort required is amortized as they can be *reused* across other designs after being written *once*. In fact, this paper considers additional network classes VGG [29], Tiny YOLO [24], and ResNet [12], where some operations (*e.g.*, parallel dot product, partial sum, bias) are *directly* reused in the designs of all network classes. The read-access operations are the most complex and the least reusable: simplifying them is aimed for in future work. An extended analysis of all operations used across all designs is provided in appendix section A.1.

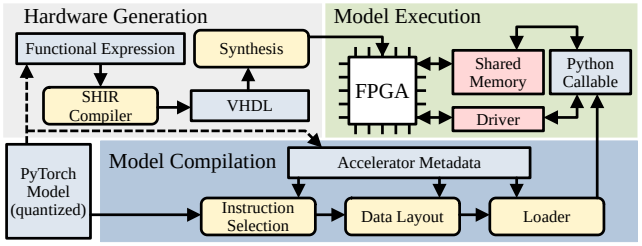


Figure 14. Compiler stack.

5 Compilation Flow

Having shown how programmable hardware is expressed functionally, we now consider the compilation flow (figure 14). A hardware designer writes a functional expression of the hardware (figure 12), which the SHIR compiler processes using rewrite and lowering rules to generate VHDL. This VHDL is then synthesized with tools like Quartus to produce an FPGA bitstream.

The hardware designer provides accelerator metadata for the model compilation step. A custom `torch.compile` [1] backend then uses this metadata to automatically map operations onto the accelerator. The metadata specifies the hardware's instruction format (for the loader), the mapping of PyTorch operations to instructions (for instruction selection), and data layout requirements (for data layout). At the end of compilation, a Python callable is produced, containing host instructions to call the FPGA driver and shared memory holding the loaded instructions and data during execution.

Note that accelerator described in section 4 assumes the sole write buffer does not overlap with any of the read buffers during a single invocation. Therefore, the compilation process ensures the allocated buffers conform to this assumption by performing liveness analysis to ensure the reuse of memory regions is done safely. In addition, since the memory accessible by the accelerator is limited by the width of specific instruction fields, additional range checking is done by the loader to ensure no overflow or truncation happens.

Finally, in the model execution phase, the Python callable uses the driver to reconfigure the FPGA (if not already done) and invokes the FPGA. The driver waits until the FPGA has fully executed the loaded instructions and eventually returns control back to the PyTorch runtime.

6 Evaluation

6.1 Experiment Setup

The proposed approach is evaluated on an Intel Arria 10 FPGA using the following four network classes: LeNet 5, VGG, Tiny YOLOv2, and ResNet. The same programmable hardware runs all networks within a class, showing multiple models can share an accelerator. The FPGA sits on a server with two Intel Xeon Gold 6254 CPUs (3.10GHz) and 1TB

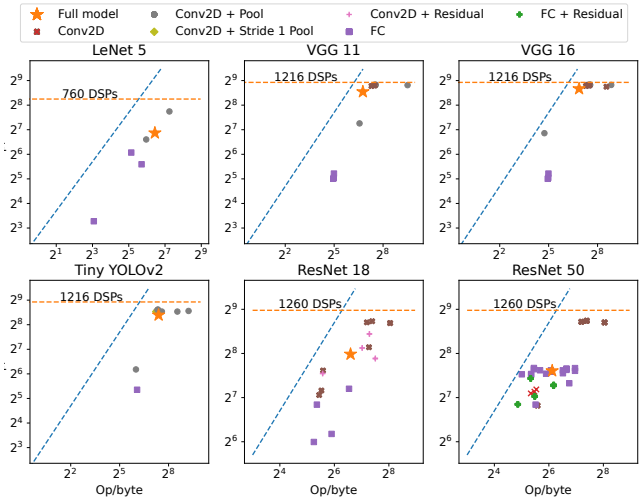


Figure 15. Roofline models of each programmable hardware. The dotted lines represents the bounds of the roofline. The stars represent the performance of the model as a whole, and the other shapes represent the performance of each layer.

RAM. Run time, measured via an on-FPGA timer, includes all host–FPGA PCI-Express transfers.

Inference Accuracy. To achieve high performance on FPGAs, accelerators typically use low-precision fixed-point values instead of floating-point. This leverages DSP blocks, which support two fixed-point multipliers. Since neural networks are usually trained in floating-point, quantizing values to fixed-point is essential.

The neural networks are trained in PyTorch on the CPU using 32-bit floating point (Fp32) and then quantized using standard techniques to 8-bit fixed point (Int8). Their floating-/fixed- point accuracies are: LeNet 5, 98.7% / 98.7%; VGG 11, 69.0% / 68.9%; VGG 16, 71.6% / 71.5%; Tiny YOLOv2, mAP 44.6 / 41.6; ResNet 18, 69.7% / 66.5%; and ResNet 50, 76.1% / 74.7%. Note that Tiny YOLOv2 uses mAP (mean Average Precision) for object detection.

6.2 Roofline Analysis

The roofline model [32] evaluates the performance of each programmable accelerator. Designs are bounded by data transfer bandwidth (6.5GB/s, forming the roofline slope) and the maximum operations per second, determined by the number of instantiated DSPs (forming the flat roof).

The roofline models of the programmable hardware designs are shown in figure 15. The star shape represents the performance of the entire model as a whole, whereas the other shapes represent the performance of different types of layers for each model. Regardless of the shape, the closer the data point is from one of the limits, the better it is.

Out of the configurations presented in figure 15, the Tiny YOLOv2 programmable hardware is the most performant as the data point for the entire network is located just under the computation limit. The VGG programmable hardware ranks next in performance. A more in-depth study of VGG 16 will be conducted in the next section.

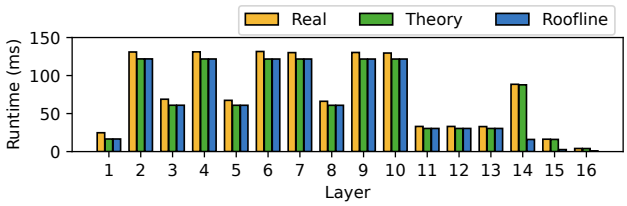


Figure 16. Per layer run time for VGG 16 (lower is better).

The LeNet 5 hardware has the lowest performance among the four network classes for two related reasons. First, its low operational intensity keeps most data near the memory bandwidth limit rather than the compute limit. Second, the small data sizes further limit bandwidth, since saturating PCI-Express requires enough data to amortize memory accesses.

ResNet networks are also a little underperforming compared to the theoretical peak. However, ResNet has a much more diverse set of layers compared to VGG or Tiny YOLOv2. Therefore, it becomes challenging to design an accelerator that achieves high performance on all layers. We will see shortly that the designs produced in this work have in fact much better performance than prior work.

6.3 VGG 16 Analysis

Figure 16 analyze VGG 16 at the layer level. Running the model on the programmable hardware measures the actual time per layer, including host data transfers. The ideal time is computed from the roofline model using each layer's operational intensity. Some layers require padding, causing under-utilization; the theoretical time accounts for this. These three values form the real, roofline, and theory bars in figure 16, with roofline $\leq$ theory $\leq$ real by definition.

For layers 1–13, the real and calculated times are similar, showing the programmable hardware is highly efficient. This aligns with figure 15, where convolution layers approach bandwidth or compute limits. Layers 14–16 (fully connected) show a noticeable gap between real and roofline times, but nearly match the theoretical time, meaning the inefficiency comes from extra padding. Most computation occurs in the first 13 layers, so slightly lower efficiency in the fully connected layers is acceptable for VGG 16.

6.4 Comparison with Prior Work

Table 2 compares VGG 16 Conv and Tiny YOLOv2 with the prior functional IR approach [14], which uses only *Let* and *FunCall* for function sharing. Since prior work [14] lacks ResNet results, the table compares ResNet against a recent hand-written accelerator [33]. The number of DSPs indicates computational parallelism. GOP/s considers both multiplication and accumulation, and DSP efficiency shows the percentage of time the DSPs are active.

Table 2. Comparison against prior works. OP counts adds and multiplies; Arria 10 FPGA supports 2 multipliers per DSP.

Benchmark	ID	Experiment	Device	Freq. (MHz)	Lat. (ms)	OP/ cycle	GOP/s.	DSP eff.	# of DSPs	ALM	Routing Cong.	
											Avg.	Peak
VGG 16 Conv	1	Prior work [14]	GX	200	68	2225	445 (1.0×)	97%	576/1518	65%	50.5%	88.5%
(Img:224x224,Int16)	2	Prior work [14]	GX	200	Not Synthesizable				1152/1518	90%	73.7%	103.5%
	3	This paper	GX	200	72	2125	425 (0.9×)	92%	576/1518	44%	31.2%	69.3%
	4	This paper	GX	200	52	2880	576 (1.3×)	63%	1152/1518	53%	43.1%	73.2%
Tiny YOLOv2	5	Prior work [14]	GX	200	13	3040	608 (1.0×)	66%	1152/1518	47%	40.8%	69.9%
(Img:416x416,Int8)	6	This paper	GX	200	11	3360	672 (1.1×)	69%	1216/1518	46%	35.1%	68.8%
ResNet 50	7	Prior work [33]	SX	170	71	664	113 (1.0×)	32%	512/1687	41%	?	?
(Img:224x224,Int8)	8	This paper	GX	200	59	681	136 (1.2×)	41%	414/1518	32%	22.5%	65.9%
	9	This paper	GX	200	45	893	179 (1.6×)	44%	510/1518	33%	23.5%	67.1%
	10	This paper	GX	200	33	1195	239 (2.1×)	43%	702/1518	34%	24.7%	66.2%
	11	This paper	GX	200	20	1945	389 (3.4×)	38%	1260/1518	36%	30.4%	68.2%

VGG 16 Conv. Prior work [14] handles VGG 16 Conv with 576 DSPs but fails to route at 1152 DSPs. In contrast, the proposed approach generates routable designs. The average routing congestion is consistently lower, allowing more DSPs to be used. The peak routing congestion (*i.e.,* the ratio of maximum routing demand to available capacity) in particular directly indicates whether a design can be routed. Values above 100% mean some FPGA regions exceed available resources, which is the case for the non-synthesizable VGG 16 experiment in prior work [14] (table 2).

The DSP efficiency is slightly lower than prior work with 576 DSPs due to the overhead of having a programmable accelerator. However, this work can exploit more DSPs, ultimately leading to a 1.3× speedup over prior work. Furthermore, this generated design is programmable and support the full VGG 16 and VGG 11 networks whereas prior work is limited to just the convolutional layers found in VGG 16.

Tiny YOLOv2. Tiny YOLOv2 is notable for its more diverse tensor sizes compared to VGG. Since prior work [14] uses fixed-size functions for sharing, it requires padding and introduces dummy elements that reduce performance. The new upper-bounded streams introduced in this paper eliminates padding, improving performance. As shown in table 2, the generated accelerator is 1.1× faster than prior work.

ResNet 50. As shown in table 2, the accelerator produced by this work is 1.6× faster than prior work [33] for similar DSP usage (510 versus 512). And again, this accelerator is fully programmable and can handle both ResNet 50 and ResNet 18 for instance. It is also possible to exploit a larger number of DSPs with modest increase in routing resources, leading to a 3.4× speedup over prior work. Prior work uses less DSPs despite having its design deployed on a device with more resources than the one used for this paper. It is likely that the prior work is unable to exploit more DSPs due to routing issues, which the approach presented in this paper avoids.

6.5 Hardware Overhead of Upper-Bounded Streams

Prior work on SHIR [26] assumes all stream lengths are known at compile time. Supporting upper-bounded streams requires extra control logic to determine the stream length at run time, mainly affecting the counters that drive handshaking, which must now handle configurable bounds.

For instance, converting a fixed *MapStm* to the upper-bounded *MapUStm* increases ALM usage (*i.e.,* use of basic logic blocks of the FPGA) from 8 to 49 for a maximum stream length of 1024. In contrast, *ReduceStm* conversion does not increase the number of ALMs required. The extra cost of *MapUStm* comes from extra logic that supports varying run-time stream length. This overhead does not occur for *ReduceUStm*, which outputs a single value and therefore has no need to memorize stream lengths.

All workloads presented require at most 129 *MapUStm* instances each. Given that the target FPGA has 427,200 ALMs, the additional ALMs used represent between 0.6%–1.5% of total resources, which is negligible.

7 Related Work

Non-Functional High-Level Synthesis. C-HLS tools such as Altera OpenCL, Xilinx Vitis HLS, and LegUp [5] provide a C-like interface for hardware design. ScaleHLS [34], SODA [19], and CIRCT [9] use LLVM or MLIR IR as intermediate abstractions. Other works [6, 30] generate synthesizable C/C++ while tuning vendor-specific pragmas such as unroll and pipeline. While these approaches improve productivity, they still require hardware expertise, and lack performance predictability compared to functional approaches [20].

Functional High-Level Synthesis. Despite higher-level abstractions, functional approaches like Chisel [3], CλaSH [2], Spatial [16], and Delite [31] still require substantial hardware expertise and manual extensions to achieve efficient accelerators and implement high-level optimizations.

Lava [4] slightly raises abstraction for HDL concepts (signals, assignments, instantiations) but remains close to traditional HDLs. μFP [28] adds higher-level support for state registers and temporal sequences, yet lacks memory and handshaking support. Prior work [8] models a valid-ready handshake protocol but lacks high-level optimizations like automatic serial-to-parallel transformation. In contrast, this paper includes instruction-style programmability, memory-related abstractions, and rewrite-based optimizations.

Aetherling [7], Lift-HLS [17], and SHIR [26] model hardware accelerators using spatial and temporal operations with vector and stream operators. However, Aetherling and Lift-HLS lack memory-related data types, limiting data reuse. Prior work on SHIR also has scalability issues due to routing congestions, as discussed in section 6.4.

AnyHLS [21] provides functional abstractions for transforming C/C++ code and uses partial evaluation to specialize and inline functions. ShakeFlow [11] models a valid-ready handshake in a functional IR for network systems. Both focus on streaming architectures and lack explicit function-level hardware sharing. Conversely, the approach in this paper supports efficient function sharing via Reduce-based sharing, upper-bounded streams, and switch-like primitive.

Resource Sharing and Optimizations. Halide-HLS [23] is not functional but optimizes line-buffer reuse for better resource utilization than Hipacc [25] on Harris corner detection. Both targets stream inputs and do not explicitly support sharing coarse-grained compute functions, as in this work.

FP-DNN [10] models memory access as a graph-coloring problem and analyzes on-chip data lifespans to reduce memory overhead. Sensei [13] evaluates the impact of bit-width post-synthesis. COSMOS [22] partitions accelerators for HLS and analyzes compute and memory bottlenecks late in the design. In contrast, this work leverages the existing SHIR [26] rewrite-based system to optimize designs with minimal extensions for upper-bounded streams.

8 Conclusion

This paper demonstrates that programmable accelerators are expressible in a purely functional IR. By introducing Reduce-based sharing, upper-bounded streams, and a lightweight switching construct, a fixed datapath controlled by a stream of instructions can be synthesized directly from high-level functional expressions. Importantly, this work builds on and integrates seamlessly with an existing functional HLS approach, requiring only minimal extensions while preserving the core abstractions. The evaluation shows that the accelerators produced for four classes of neural networks achieve high performance. In addition, the produced accelerators are able to exploit more DSPs than prior work leading to speedups between 1.1×–3.4×.

Acknowledgments

We thank Christof Schlaak for designing and implementing the core SHIR project, which served as the compiler backend for this work. This work was supported by the Fonds de Recherche du Québec–Nature et Technologies' 3rd cycle scholarship, award #346530. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889] and [grant RGPIN-2026-06336], and the Canada CIFAR AI Chairs Program.

A Appendix

A.1 Programmable Designs for Other Networks

The approach shown in section 4 is also used to express the neural networks of VGG [29], Tiny YOLOv2 [24], and ResNet [12]. The programmable hardware for VGG, Tiny YOLOv2, and ResNet follow an overall structure similar to figure 12: all four designs perform a reduction that writes to host memory (lines 1–2) over an upper-bounded stream of data (lines 4–12) and write address pairs (line 13). The difference between these four designs lie in the generation of the upper-bounded stream being reduced, which in turn, are dictated by the structure and the operations of the neural network. The full list of operations, along with their usage per design, are summarized in table 3. As before, these operations are divided into three classes: compute, read access, and write access.

Compute. All neural networks surveyed in this work perform parallel dot products, partial sums, requantization, and adding biases to either the convolution or the fully connection layer. Thus, it is sufficient to write the SHIR expression for each of these functional units *once* and *reuse* them across all designs. Not all neural networks use the same activation and pooling operation. For example, Tiny YOLOv2 is the only neural network that makes use of the Leaky ReLU activation function and a 2D max pooling with stride 1. Thus, only the Tiny YOLOv2 design includes the SHIR expressions for these two operations. The programmable hardware for ResNet also requires additional logic to handle residual connections that occur in a subset of layers.

Read Access and Write Access. The data access operations include reading the images, weights, biases; buffering into on chip memory as necessary; and generating the write addresses for the neural network output. While all neural networks also make use of these functionality, due to the shape of the data (*i.e.,* how wide the input and output channel is, and thus, how well the cache line is utilized), these components are further specialized into shallow and deep data access components to eliminate waste during data transfer between the programmable hardware and the host memory.

Table 3. A summary of the size of the operations, in terms of IR nodes, and the amount of times each operation is present in each programmable hardware design.

Operation	Size	LeNet	VGG	YOLO	ResNet
Instruction Reduction					
Read	26	1	1	1	1
Reduction	1	1	1	1	1
Compute					
Parallel dot product	34	1	1	1	1
Partial sum	26	3	4	4	2
Bias add	10	2	1	1	1
Residual add	26	0	0	0	1
Requant rescale	18	1	1	1	1
Activation ReLU	1	1	1	1	1
Activation leaky ReLU	8	0	0	1	0
Pooling max	39	0	1	1	1
Pooling max stride 1	97	0	0	1	0
Pooling average	42	1	0	0	0
Read Access					
Conv image address	58	0	1	1	3
Conv image tile indexing	29	0	1	1	0
Conv image read	56	0	1	1	1
Conv image buffer	23	0	1	1	1
Conv weight address	47	0	1	1	2
Conv weight read	29	0	1	1	1
Conv weight buffer	61	0	1	1	1
Conv image (shallow)	39	1	0	0	0
Conv weight (shallow)	39	1	0	0	0
FC image (shallow)	49	1	0	0	0
FC weight (shallow)	60	1	0	0	0
Bias read	47	1	1	1	1
Residual read	71	0	0	0	1
Requant read	53	1	1	1	1
Write Access					
Reshape pool	47	0	1	1	1
Reshape stride 1 pool	46	0	0	1	0
Output address	39	0	1	1	1
Output write	25	0	1	1	1
Output (shallow)	13	1	0	0	0
Total		18	23	26	25

Data-Availability Statement

This paper's artifact is available on Zenodo [15], including the Docker image and all other required software, except for the licensed Intel/Altera tools and the Arria 10 FPGA PAC board.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[2] C.P.R. Baaij. 2015. *Digital circuit in CλaSH: functional specifications and type-directed synthesis*. Ph. D. Dissertation. University of Twente, Netherlands. doi:10.3990/1.9789036538039 eemcs-eprint-23939.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. doi:10.1145/2228360.2228584

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: hardware design in Haskell. *Acm Sigplan Notices* 34, 1 (1998), 174–184. doi:10.1145/291251.289440

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. doi:10.1145/1950413.1950423

[6] Young-kyu Choi and Jason Cong. 2018. HLS-based optimization and design space exploration for applications with variable loop bounds. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. doi:10.1145/3240765.3240815

[7] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3385412.3385983

[8] Stephen A Edwards, Richard Townsend, Martha Barker, and Martha A Kim. 2019. Compositional dataflow circuits. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 1 (2019), 1–27. doi:10.1145/3274280

[9] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. 2021. MLIR as hardware compiler infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*. https://woset-workshop.github.io/PDFs/2021/a06.pdf

[10] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 152–159. doi:10.1109/FCCM.2017.25

[11] Sungsoo Han, Minseong Jang, and Jeehoon Kang. 2023. ShakeFlow: Functional hardware description with latency-insensitive interface combinators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 702–717. doi:10.1145/3575693.3575701

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. doi:10.1109/CVPR.2016.90

[13] Hsuan Hsiao and Jason H Anderson. 2018. Sensei: An area-reduction advisor for FPGA high-level synthesis. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. doi:10.23919/DATE.2018.8341974

[14] Tzung-Han Juang, Christof Schlaak, and Christophe Dubach. 2023. Let Coarse-Grained Resources Be Shared: Mapping Entire Neural Networks on FPGAs. *ACM Trans. Embed. Comput. Syst.* 22, 5s, Article 114. doi:10.1145/3609109

[15] Tzung-Han Juang, Paul Teng, and Christophe Dubach. 2026. A Functional Approach to Synthesizing Routable Programmable Accelerators for Neural Networks. Zenodo. doi:10.5281/zenodo.20046007

[16] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/3192366.3192379

[17] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. 2019. High-level Synthesis of Functional Patterns with Lift. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY)*. doi:10.1145/3315454.3329957

[18] Y. LeCun. 1989. Generalization and Network Design Strategies. In *Connectionism in Perspective*, R. Pfeifer, Z. Schreter, F. Fogelman, and L. Steels (Eds.). Elsevier, Zurich, Switzerland. http://yann.lecun.com/exdb/publis/pdf/lecun-89.pdf an extended version was published as a technical report of the University of Toronto.

[19] Marco Minutoli, Vito Giovanni Castellana, Cheng Tan, Joseph Manzano, Vinay Amatya, Antonino Tumeo, David Brooks, and Gu-Yeon Wei. 2020. Soda: a new synthesis infrastructure for agile hardware design of machine learning accelerators. In *Proceedings of the 39th International Conference on Computer-Aided Design*. doi:10.1145/3400302.3415781

[20] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/3395657

[21] M Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. 2020. AnyHLS: High-level synthesis with partial evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11. doi:10.1109/TCAD.2020.3012172

[22] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. 2017. COSMOS: Coordination of high-level synthesis and memory optimization for hardware accelerators. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s. doi:10.1145/3126566

[23] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3. doi:10.1145/3107953

[24] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: Better, Faster, Stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6517–6525. doi:10.1109/CVPR.2017.690

[25] Oliver Reiche, M. Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. 2017. Generating FPGA-based image processing accelerators with Hipacc: (Invited paper). In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1026–1033. doi:10.1109/ICCAD.2017.8203894

[26] Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. 2022. Memory-Aware Functional IR for Higher-Level Synthesis of Accelerators. *ACM Trans. Archit. Code Optim.* 19, 2, Article 16. doi:10.1145/3501768

[27] Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. 2022. Optimizing Data Reshaping Operations in Functional IRs for High-Level Synthesis. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2022)*. doi:10.1145/3519941.3535069

[28] Mary Sheeran. 1984. muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. 104–112. doi:10.1145/800055.802026

[29] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. doi:10.1109/ACPR.2015.7486599

[30] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4. doi:10.1145/3494534

[31] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 134. doi:10.1145/2584665

[32] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. doi:10.1145/1498765.1498785

[33] Xiaoru Xie, Jun Lin, Zhongfeng Wang, and Jinghe Wei. 2021. An efficient and flexible accelerator design for sparse convolutional neural networks. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 7 (2021), 2936–2949. doi:10.1109/TCSI.2021.3074300

[34] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. doi:10.1109/HPCA53966.2022.00060

Received 20 March 2026; accepted 1 May 2026