Village Synthesis

Liam Strand
Northwestern University
Evanston, IL, USA
ltrs@u.northwestern.edu

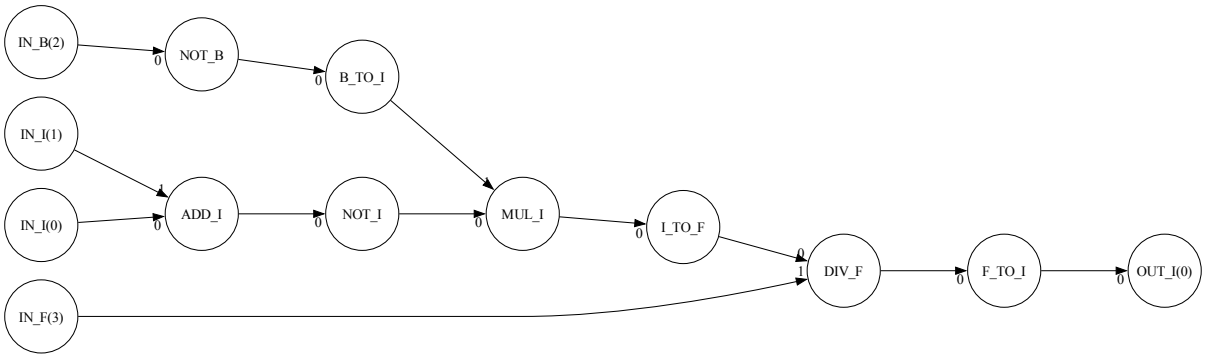

Figure 1: The abstract representation of a VIR operator dataflow subgraph that can now be lowered to hardware.

ABSTRACT

Programming specialized hardware accelerators like FPGAs is notoriously difficult, creating a gap between high-level algorithmic expression and hardware implementation. This work bridges that gap with a toolchain for compiling the nested data-parallel language NESL directly to hardware. We leverage the existing front-end of the Village compiler to transform a high-level program into a dataflow graph representation, which we then treat as a blueprint for a hardware accelerator. By mapping graph nodes to a library of Chisel functional units, including multi-cycle floating point operators, and automatically wiring them together, our system automates the hardware design process. This research demonstrates the feasibility of a "push-button" approach to hardware generation, enabling programmers to leverage custom accelerators in the Village ecosystem without manual hardware design.

KEYWORDS

Heterogeneity, FPGA, HDL Synthesis, Nested-Data Parallelism

1 INTRODUCTION

The relentless pursuit of greater computational power has driven the high-performance computing landscape towards heterogeneity, where systems are increasingly composed of diverse computational resources like GPUs, TPUs [14], and FPGAs. While this hardware specialization offers immense potential, it creates a significant programmability crisis, as developers must master disparate, low-level programming models for each device. High-level synthesis [10] has emerged as a critical technology to address this challenge, promising to abstract away the complexities of hardware design by automatically generating specialized circuits from high-level algorithmic descriptions.

Motivation. Unfortunately, high-level synthesis tools struggle to map from the C-style languages that software engineers are familiar with to efficient hardware implementations [12]. With modern tools such as AMD's Vitis, however, it is possible to achieve the desired hardware design. But, actually generating an efficient design requires extensive refactoring, often to the point where the C or C++ code mirrors what would appear in a hardware description language like Verilog [2].

Is it possible to take advantage of the semantic richness of a high-level parallel language to generate an efficient custom hardware design? Can this generation be done transparently to the end-programmer?

Contributions. We have leveraged the Village compiler pipeline (described in Section 2.1) and the Chisel hardware description language to create a toolchain to synthesize high-level programs written in NESL to hardware designs. Our contributions are as follows:

- We present a novel synthesis backend that lowers the Village Intermediate Representation (VIR), a high-level dataflow graph, into a custom hardware design using Chisel.
- We design and implement a library of reusable hardware functional units in Chisel that correspond to VIR instructions. This library features decoupled interfaces for asynchronous operation and integrated operand buffering.
- We demonstrate the end-to-end correctness of our synthesis pipeline through simulation-based testing on a variety of VIR graphs, validating the translation from the abstract dataflow representation to a functional hardware circuit.

Methodology. This work makes extensive use of the Chisel hardware description language (described in Section 2.2). Testing is performed under simulation using Verilator, driven by the `scalatest` library. Simulation is architecture independent and the testing suite has been run on both ARM and x64 machines. The source code for this work is available at https://github.com/vlang-project/village-synthesis.

Limitations. Integration into the Village compiler and runtime has not yet been completed, so it is not yet possible to fully compile a NESL program to hardware by invoking a single command. Additionally, the hardware functional units that have been built for this work have not been manually optimized or internally pipelined. Finally, the hardware designs have not yet been run on actual hardware, instead opting for simulation-based testing to stay within time constraints.

2 BACKGROUND

2.1 Village

Implementing complex algorithms in low-level parallel languages like C or C++ with CUDA [18], OpenMP [11], or MPI [16] is challenging. These languages force the programmer to specify precisely how their algorithm should be mapped onto hardware. CUDA is especially troublesome in this regard, exposing every minute facet of a GPU's architecture. A programmer may choose to ignore those details and write a direct implementation. But, they sacrifice performance compared to a programmer who is willing to optimize to the GPU's microarchitecture and write complex, inscrutable code that obfuscates the underlying algorithm.

The other options available to programmers are high-level parallel languages. Languages like Chapel [8], Bend [21], or NESL [6, 7] can ergonomically express parallel algorithms, and their rich semantics provide an intelligent compiler multitudinous opportunities to automatically optimize the implementation. Functional high-level parallel languages are even able to abstract away the execution model. Unfortunately, these benefits often come at the cost of performance. High-level programming languages can struggle to be as performant as their low-level counterparts because the programmer does not have direct control of hardware resources, making extensive optimization essentially impossible.

The Village project [17] provides a new implementation of NESL that aims to preserve its expressive high-level parallel semantics while generating code that has the performance of a low-level parallel language. The compiler produces a static executable by lowering NESL to VCODE (Section 2.1.2), then VCODE to VIR (Section 2.1.3), VIR to LLVM-IR [15], and finally relying on LLVM's backend to emit machine code. In this work, we aim to intercept the compilation process at the VIR stage, and synthesize parts of the program to a custom hardware design.

2.1.1 NESL. Designed in 1992 and implemented in 1996, NESL [6, 7] is a high-level parallel language that uses the nested data parallelism model of computation. This model allows programmers to express the parallelism of their algorithms in terms of applying a function in parallel to a set of values, where those values could themselves be parallel functions. It is very flexible and allows the programmer to express all of the parallelism available in their algorithm.

```
1  function qsort(a) =
2    if (#a < 2) then a
3    else let
4      pivot   = a[#a/2];
5      lesser  = {e in a | e < pivot};
6      equal   = {e in a | e == pivot};
7      greater = {e in a | e > pivot};
8      result  = {qsort(v) : v in [lesser, greater]}
9    in result[0] ++ equal ++ result[1];
```

Figure 2: A NESL implementation of quicksort, exposing the parallelism of the scan over the collection `a` and the parallel recursive calls to `qsort`.

Essentially, NESL is a version of Standard ML that has been extended to support parallel semantics. It feathers a parallel map/-filter operator and collection types for that operator to consume. Recursion semantics are also updated to allow for parallel recursive calls.

Figure 2 contains the NESL implementation of quicksort, a classic parallel algorithm. Lines 5-7 express flat data parallelism, with each line applying a filter to the collection. Line 8 shows the recursive calls over the two subcollections, which can be executed in parallel. A flat data parallel language would struggle to reveal the parallelism available in quicksort so succinctly and clearly.

2.1.2 VCODE. Since NESL would be far too complex to execute directly, it is first lowered to a simpler intermediate representation. VCODE uses a relatively small set of instructions that operate on a stack of segmented vectors, which helps simplify an implementation. This translation process maps nested parallelism to equivalent flat parallelism. This means that parallel recursive calls to parts of a collection, as can be seen in line 8 of Figure 2, are reduced to a single call over a segmented vector. Although this flattening simplifies the implementation of a VCODE interpreter, it is still able to represent nearly all of the parallelism available in the original NESL program.

Translating a stack-based intermediate representation such as VCODE into a hardware design would be a monumental task. Fortunately, prior work allows VCODE to be translated to another intermediate representation that is much more amenable to lowering to hardware.

2.1.3 VIR. Village Intermediate Representation (VIR) is a vector dataflow language that uses simple instructions nearly identical to those of VCODE. This makes it an excellent representation of a program to lower to hardware. Instead of a stack of vectors with global state, VIR is simply a dataflow graph connecting inputs to outputs. The textual syntax of VIR is represented as a sequential single-static-assignment program, but the backing representation is of a true dependency graph between operators.

A dataflow graph representation is very helpful for hardware synthesis because it makes the dependencies between operations

```
1  (%V1) = vcode_not_bool(%A2)
2  (%V2) = vcode_add_int(%A0, %A1)
3  (%V3) = vcode_bool_to_int(%V1)
4  (%V4) = vcode_not_int(%V2)
5  (%V5) = vcode_mul_int(%V3, %V4)
6  (%V6) = vcode_int_to_float(%V5)
7  (%V7) = vcode_div_float(%V6, %A3)
8  (%V8) = vcode_float_to_int(%V7)
```

Figure 3: A simple VIR program that includes all three datatypes, and unary and binary operators.

explicit. In hardware design, computations are performed by physical functional units which are wired together. Serving as a direct blueprint for these connections, a dataflow graph maps how the output of one operation feeds into the input of another. This explicit representation of data movement is much easier to translate into a physical circuit than a stack-based representation like VCODE, where dependencies are implicit in the order of push and pop operations on a shared stack.

Figure 3 contains many operations that are interdependent. We see, for example, that `vcode_div_float` on line 7 must wait for `vcode_int_to_float` on line 6, which must in tern wait for its parent, and eventually `vcode_not_bool` and `vcode_add_int` on lines 1 and 2, whose operands are supplied immediately as arguments. For a graphical representation of these dependencies, see Figure 1.

VIR is a representation that is significantly closer to the physical structure of a hardware circuit, which greatly simplifies the task of synthesizing the high-level algorithm into hardware.

2.2 Chisel

To perform the synthesis, we rely on Chisel [5], a new hardware description language embedded in Scala [19]. Chisel code takes the form of *generators*, which are Scala classes that can be invoked by the Chisel compiler to generate a specific instance of a hardware module. Generators can take parameters and are highly composable using both object-oriented and functional programming paradigms.

During compilation, the Scala code is executed to generate the circuit in an intermediate representation called FIRRTL (Flexible Intermediate Representation for RTL). This FIRRTL is then passed to a separate compiler which performs hardware-specific optimizations before emitting a final, low-level hardware description such as Verilog [1].

Chisel has been used to build many complex hardware projects, including high-performance RISC-V cores like the in-order Rocket [4] and the out-of-order SonicBOOM [23]. Moreover, Chisel is the foundation of the Chipyard [3] framework for SoC design, simulation, and synthesis.

In this work, Chisel is used to map VIR nodes to hardware functional units, wire up the connections between those functional units, and wrap the graph of functional units in an interface that can be accessed as a discrete accelerator from the outside world.

```
sealed trait VillageOperand
case object BOOL extends VillageOperand
case object INT extends VillageOperand
case object FLOAT extends VillageOperand

sealed trait VcodeNode {
    val inputs:  Seq[VillageOperand]
    val outputs: Seq[VillageOperand]
    val id: Int = VcodeNode.next()
}
```

Figure 4: The `VcodeNode` type that is used to internally represent a VIR graph within Scala.

```
sealed case class ADD_I() extends VcodeNode {
    val inputs = Seq(INT, INT);
    val outputs = Seq(INT)
}
```

Figure 5: A simple binary operator node class.

3 OPERATOR GRAPHS IN SCALA

In order to transform a VIR graph into a hardware design, we must first bring the VIR graph into the Chisel ecosystem.

3.1 Node Contents

Each VIR node is represented as a subclass of a `VcodeNode`, which provides them with a globally unique identifier and requires that the node specifies its I/O (see Figure 4). An addition, for example, takes two integer inputs and produces one integer output. This interface described in Scala as in Figure 5.

To interact with the outside world, the graph must define where inputs are supplied and outputs are read. I/O nodes serve this purpose. From the perspective of the `VcodeGraph`, an input node produces a single output of the specified type and an output node consumes a single input of the specified type. Each I/O node is also assigned an index number to specify the order in which the inputs and outputs appear when combined in the accelerator's I/O.

This complex object-oriented-class-hierarchy-mixin preamble is necessary to allow future generation steps to pattern match on the type of node to dispatch to the correct hardware generator.

3.2 `scala-graph` DSL

We take advantage of the full-featured `scala-graph` library [9] to represent the edges between nodes. This library allows us to create a typed graph of `VcodeNodes` that are linked by labeled, directed edges. Some VIR operations are not commutative, so edges must be labeled to indicate argument order.

These graphs can be imported and exported as JSON, and they are easily queryable and transformable from within Scala.

As a bonus, `scala-graph` also uses Scala implicits to create a domain-specific language (DSL) for describing these VIR graphs. Figure 6 shows an example of this DSL in-use. The `~>` operator

```
val add = new ADD_I
val graph = VcodeGraph.from(Seq(
    new INPUT_I(0) ~> add :+ 0,
    new INPUT_I(1) ~> add :+ 1,
    add ~> new OUTPUT_B(0) :+ 0
))
```

Figure 6: A small `VcodeGraph` that adds two numbers together using the `scala-graph` DSL.

denotes a directed edge and the `:+` operator assigns the edge label. The DSL is helpful when defining graphs for unit testing, or for quick compilation and synthesis checks.

3.3 Graph Validation

This representation permits any layout of nodes, with any set of edges connecting them. Many graphs can be represented but which cannot be synthesized to hardware. Some examples include cycles, disconnected graphs, and graphs with duplicate edge labels. Before we can begin the synthesis process, we must validate that the graph is, in fact, synthesizable. The properties asserted are as follows:

- The graph is acyclic.
- The graph is connected.
- Each node has the same number of incoming and outgoing edges as it has logical inputs and outputs.
- Each node's incoming edges are numbered in increasing order starting at 0.
- For each edge, the output type of its source is the same as the input type at its target.
- The graph's input and output nodes are numbered in increasing order starting at 0.

Two properties follow from these assertions. First, the set of input nodes and the set of output nodes can each be joined into a continuous sequence. Second, every node is connected to the correct number of inputs and outputs that will supply and consume values of the correct type.

4 HARDWARE FUNCTIONAL UNIT LIBRARY

Each node in the `VcodeGraph` needs to be mapped to a hardware module. Many of these modules are quite simple, needing only a few lines of Chisel code. Others, specifically floating point operations, are much more complex.

This complexity is hidden behind a shared interface that is implemented by all functional units. The interface, shown in Figure 7, specifies that all inputs and outputs are *decoupled*, meaning that the producer must mark the data as `valid` and the consumer must be `ready` to receive it before the transfer takes place. Moreover, it places all inputs in a single `Vec` that can be indexed into, and limits each functional unit to a single output. Removing this limitation is a work-in-progress.

Each functional unit extends the `VillageFunctionalUnit` class and uses I/O provided by one of the `HasVillageIO` specializations (see Section 5.1).

```
trait HasVillageIO {
    val io: Bundle {
        val in: Vec[DecoupledIO[UInt]];
        val out: DecoupledIO[UInt];
    }
}
abstract class VillageFunctionalUnit(n: VcodeNode)
    extends Module with HasVillageIO
    { /* bookkeeping */ }
```

Figure 7: The interface shared between all functional units.

```
1 val functional_units = Seq(
2   new VillageUnaryFU(new NOT_I(),  ~_),
3   new VillageBinaryFU(new ADD_I(), _+_),
4   new VillageIO(new INPUT_I(0)),
5   new VillageFloatAddFU(new ADD_F()),
6   new VillageFloatCmpFU(new EQ_F(), EQ),
7   new VillageFloatToIntFU(new TRUNC_F(), TRUNC)
8 )
```

Figure 8: Instantiating several different functional units.

4.1 Simple Unary and Binary Functional Units

Most VIR instructions are simple unary and binary arithmetic operations. These functional units are small enough that they can execute in a single cycle. Once their operands become `valid`, they can immediately produce a `valid` result.

This simplification allows the Scala generator to be parameterized over the operation that needs to be performed. The same generator can be used for an integer addition, bitwise XOR, or boolean equality. In lines 2 and 3 of Figure 8, we see the generators used to create a `NOT_I` and an `ADD_I` functional unit. Of the 42 functional units defined in this work, 21 use either the `VillageBinaryFU` generator or the `VillageUnaryFU` generator.

Since internal I/O is delegated to `HasVillageIO` specializations, the actual implementation of each of these generators requires only six lines of code. Moreover, the implementation of a functional unit using one of these generators requires a single line of code as shown in Figure 8.

4.2 Input and Output

Input functional units receive their input from the outside world and provide output to functional units that perform computations. Since all connections between functional units are 64 bits wide (allowing for wider busses is an area of active development), boolean inputs are first or-reduced so that the integer value of the wire is either 0 or 1.

On the output side, functional units receive their input from another functional unit and provide output to the outside world. Output functional units also or-reduce boolean values before emitting them.

The outside world sees these nodes through the accelerator interface, which is described in Section 5.4.

4.3 Floating Point with `hardfloat`

Floating point functional units are by far the most complicated in this system. While floating point addition can be completed in a single cycle, division and square root computations can span many cycles. Moreover, implementing a high-performance floating point computation in hardware is difficult. Fortunately, the `hardfloat` library [22] provides high-quality implementations of all basic floating point operations.

The operands passed into `hardfloat` modules use a recoded version of the IEEE 754 standard [13], so each floating point functional unit first converts its inputs from IEEE 754 format to the recoded format before passing the bits off to the appropriate `hardfloat` module. When ready to emit a result, the output is then reformatted back to adhere to the IEEE 754 standard. This way, all floating point values in wires or in visible registers are in IEEE 754 format. Since the reformatting operation is a simple matter of bit swizzling, the overhead is negligible.

There are additional challenges to consider when designing functional units for multicycle computations like division or square root. The `hardfloat` modules signal when they are `ready` for input and when their outputs are `valid`. These signals need to be wired to the functional unit I/O so that the `hardfloat` module does not get swamped with operands. The `hardfloat` division module specifically supports some overlapping of operands, but we are unfortunately unable to take advantage of this feature at this time.

Floating point comparisons are challenging because `hardfloat` provides one "total-comparison" module rather than a separate module for each type of comparison (greater-than, equal, etc). Luckily, this issue can be addressed with Chisel features. The floating point comparison generator takes a parameter that selects the comparison to perform. In line 6 of Figure 8, the generator is passed the `EQ` flag to emit a functional unit that checks for floating point equality.

4.4 Conversions

Finally, there is at the last category of functional unit: type conversions. We may want to convert an integer into a boolean, or round a floating point value into an integer. Converting between integers and booleans is straightforward. To transform a boolean into an integer we emit `1` if the input is nonzero and `0` otherwise. Casting an integer to a boolean is identical.

Conversions to and from floating point values are, as usual, more complex. The modules provided by `hardfloat` to perform these conversions both accept as input a "rounding mode" signal. For conversions from integers to floats, the module is always configured to round to the nearest representable number. There are three separate VIR instructions for rounding from a float to an integer: ceiling, floor, and truncation (round towards zero). Each of these instructions is mapped to a functional unit that is preconfigured to perform the appropriate rounding operation. An instantiation of the functional unit for `TRUNC_F` is shown in line 7 of Figure 8.

5 CONNECTING FUNCTIONAL UNITS

With VIR nodes mapped to functional unit generators, we can generate a functional unit for each of the nodes in our graph. The functional units are stored in a hash table using the associated node ID as a key. Each functional unit also stores a reference to the node that generated it. This way, it is trivial to traverse the tree of functional units by using the metadata stored in the associated node.

```
trait HasVillageBinaryIO extends HasVillageIO {
  override val io = IO(new Bundle() {
    val in = Flipped(Vec(2,
        Decoupled(UInt(64.W))))
    val out = Decoupled(UInt(64.W))
  })
  val in_a = Queue(io.in(0), 4, false, true)
  val in_b = Queue(io.in(1), 4, false, true)
  in_a.nodeq(); in_b.nodeq(); io.out.noenq()
}
```

Figure 9: A specialization of `HasVillageIO` that buffers two input operands.

5.1 `HasVillageIO` Specializations

While it would be possible to define a new I/O bundle that implements `HasVillageIO` for every functional unit generator, it is more straightforward to build mix-in traits for the two cases (unary and binary) and reuse them throughout the functional unit library. The output bundle is identical in each case, using a single decoupled 64-bit integer. `HasVillageUnaryIO`'s input bundle contains a `Seq` with a single decoupled 64-bit integer. The binary version's input is the same except it has two such integers as shown in Figure 9.

Note that the inputs are individually decoupled, meaning that the functional unit can indicate, for example, that it is ready to receive an input on connection 0 but not on connection 1.

5.2 Operand Buffering

In a dataflow graph such as the one shown in Figure 1, it is clear that some functional units may receive each of their operands during different cycles. For example, the `DIV_F` node will almost certainly receive operand 1 (the denominator) well before it receives operand 0 (the numerator). We need a way to store the earlier operand and wait for the latter operand before proceeding.

To accomplish this, `HasVillageIO` specializations implement operand queuing and buffering as shown in Figure 9. Each functional unit can store a globally configurable number of operands (four by default) for each of its inputs. Instead of reading directly from the I/O bundle, the functional units read their inputs from the consumer side of the operand queues.

This adds a single cycle of latency to each of the connections, which is a significant performance penalty, especially when considering that individual functional units can often execute in a single cycle. In exchange, it becomes possible to synthesize graphs where a functional unit may have its operands arrive during different cycles. Moreover, if a functional unit takes a variable number of cycles to execute, it can now buffer operands so that the whole accelerator is not stalled if a single operation takes modestly longer than expected.

```
for (e <- vir_graph.edges) {
  val source = functional_units(e.source.id)
  val target = functional_units(e.target.id)
  target.io.in(e.label) <> source.io.out
}
```

Figure 10: Connecting all functional units together.

The performance penalty of this buffering is mitigated by combinatorially connecting the `valid` signals from the input bundle to the consumer end of the buffer queue. This allows the input to be read one cycle sooner than would otherwise be possible.

5.3 Wiring It All Up

After instantiating the functional units, we must connect them together following the edges in the VIR instruction graph. Each input and output is a decoupled interface, so in addition to the data connections, we must also pass a parallel `valid` wire and a reversed `ready` wire. Thus, a unary functional unit requires six connections; receiving data and a `valid` signal from its predecessor and a `ready` signal from its successor, and sending data and a `valid` signal to its successor and a `ready` signal to its predecessor.

The complex task of managing all of these connections is vastly simplified by Chisel's <> operator, which connects all of the outputs from one side to matching inputs on the other side, and vice versa. The Scala type system ensures that the two modules have compatible I/O. Since every functional unit is required to implement `HasVillageIO`, the type system can guarantee that `io.in(0)`, a flipped (i.e., swapped inputs and outputs) decoupled 64-bit integer, can connect to `io.out`, a decoupled 64-bit integer.

So, after carefully constructing a type hierarchy to enforce compatible interfaces between functional units, we are able to wire up the entire graph using four lines of code, shown in Figure 10.

5.4 Accelerator I/O

A connected graph of functional units is useless unless it has an interface to communicate with the outside world. This interface is asynchronous and ordered. All elements of an input are supplied at once whereupon the client of the accelerator must wait for the output. For the graph shown in Figure 1, a client would supply four 64-bit numbers: two integers, a boolean extended to 64 bits, and a 64-bit IEEE 754 floating point number.

The inputs and outputs are decoupled *as a collection*. There is no notion of partial validity or readiness from outside of the accelerator. Therefore, the accelerator is either `ready` to receive inputs or it is not. Likewise, its outputs are either `valid` or not. The client of the accelerator must also mark their inputs as entirely `valid` or not, and signal the accelerator if they are `ready` for all output.

The unified interface is far more straightforward for any programmer or driver developer who intends to interact with the device.

6 RESULTS

Due to time constraints, the accelerator synthesis system has not been integrated into the Village compiler and runtime. Thus, the results presented here demonstrate the correctness of the graph representation, functional unit library, and I/O system rather than real-world performance metrics.

6.1 Testing Methodology

To minimize debugging surface area, tests are run in simulation using Verilator [20]. Tests utilize the `scala-graph` DSL to define small operator graphs. Examples include a three-way addition constructed from two adders, or a tree of several boolean logic operations.

These graphs are then passed through the synthesis pipeline, producing a testable accelerator. The accelerator is then supplied `valid` inputs, and the clock is stepped until the outputs are `valid`. At that point, values are read from the output of the accelerator and compared to values calculated in software.

6.2 Testing Results

6.2.1 Three- and Five-Way Adders. These tests evaluate the accelerator's ability to handle buffering operands. The final adder must wait for all previous computations to complete while also being supplied an input that is ready immediately. The early operand must be stored until the other is ready.

6.2.2 Sequenced Inputs. These tests force the accelerator to process multiple sets of inputs. In one set of tests, inputs are supplied one at a time and the testing harness waits for an output to be produced before supplying the next input. This ensures that the accelerator resets itself when it has exhausted all of its inputs.

In another test suite, all inputs are supplied before any outputs are read. This tests the capacity of the internal buffers and ensures that the accelerator remains `ready` for input for as long as possible.

6.2.3 Integer Arithmetic and Bitwise Logic. These tests check that all integer arithmetic and logical units behave as expected. Special attention is given to checking subtle overflow, underflow, and bit-shifting behaviors.

6.2.4 Boolean Logic. These tests replicate the integer bitwise logic tests and verify that the boolean versions of those functional units behave as expected.

6.2.5 Floating Point Arithmetic. These tests ensure that modules from the `hardfloat` library have been integrated correctly into functional units in our ecosystem. Care is taken to ensure that values are converted to and from `hardfloat`'s recoded format correctly.

6.2.6 Comparisons. These tests compare between positive numbers, negative numbers, and zero in both integer and floating point representations. Integer comparisons must be handled as signed two's-compliment comparisons.

7 FUTURE WORK

While this work allows hardware accelerators to be generated from abstract VIR instruction graphs, more engineering is needed to integrate and optimize these accelerators.

7.1 Simulated Integration

To prototype the software interface between the accelerator and the Village runtime without needing to debug FPGA issues, an accelerator simulator library can be generated by Verilator. Then, the runtime can use simple function calls to communicate with the simulated accelerator. Then, a simulated version of the accelerator can be interacted with through simple function calls into this generated library. This would allow developers to quickly iterate on the runtime-facing API that will eventually be implemented by a custom hardware driver.

7.2 Hardware Integration

Once the API has stabilized, the next task is to connect the accelerator to the FPGA's physical PCIe interface [1] and write the driver required to communicate through that interface. Since the hardware design, driver implementation, and runtime software are all designed to collaborate, the resulting communication channel is tailored specifically for the accelerator. This tight integration simplifies driver development, minimizes communication overhead, and eliminates common bugs that arise from mismatches between hardware and software interface specifications.

7.3 Superscalar

A single accelerator is highly unlikely to require all of the space available on an FPGA. It should be simple to replicate the accelerator many times to utilize the available area. Then, an arbiter can be added to dispatch inputs to the available graph accelerators in a round-robin manner. On the other side, outputs could be read into a shared buffer and shipped off the FPGA. With multiple accelerator instances operating independently, the system can process several distinct computations in parallel, much like a multi-core processor handling different threads. As soon as one accelerator begins processing a task, the arbiter can dispatch the next task to an idle instance, enabling a deep pipeline of concurrent execution that dramatically boosts the system's total throughput.

7.4 Dynamics

So far, we have focused on statically determining what part of a VIR operator graph should be synthesized to hardware at compile-time. The Village compiler performs profile-guided optimizations, so it can make an informed guess about what subgraphs are particularly amenable to synthesis. But, if the runtime behavior of the program varies drastically from the profiled behavior, it would be very powerful to be able to dynamically reconfigure which parts of the program are executed on the custom hardware and which remain in software.

8 CONCLUSION

This work presents a novel approach to compiling parallel algorithms written in NESL directly to hardware. We have built a synthesis pipeline that brings a VIR graph into Chisel, maps its nodes to hardware functional units, connects those functional units together, and wraps the whole graph in a convenient interface. Our implementation leverages the Scala type system to encode the VIR graph in a format that is amenable to synthesis and enforce consistent interfaces between functional units. Extensive unit testing has demonstrated that the translation from VIR to hardware is performed accurately, regardless of the type of instructions or size of the graph. This research is a substantial first step towards integrating dynamically-generated custom accelerators into the Village ecosystem.

[1] We are currently targeting a BittWare XUP-P3R.

ACKNOWLEDGMENTS

The author thanks the following people for their contributions to this work and paper:

- Jacqueline Egidio, Brad Strand, and Kristina Strand for suggestions to improve the clarity of this manuscript.
- Karl Hallsby for his frequent and thoughtful help understanding both Scala's and Chisel's extraordinary compiler and synthesis errors.
- Prof. Peter Dinda for his guidance and encouragement throughout this project.

REFERENCES

[1] 2024. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. (2024), 1–1354. https://doi.org/10.1109/IEEESTD.2024.10458102

[2] Advanced Micro Devices, Inc. 2024. *HLS Optimization Methodologies.* Advanced Micro Devices, Inc. https://docs.amd.com/r/en-US/ug1399-vitis-hls/HLS-Optimization-Methodologies Document UG1399 (v2024.1). Accessed: June 12, 2025.

[3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[4] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[5] 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* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, 1216–1225. https://doi.org/10.1145/2228360.2228584

[6] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan Hardwick, Jay Sipelstein, and Marco Zagha. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel and Distrib. Comput.* 21, 1 (April 1994), 4–14.

[7] Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *Proceedings of the International Conference on Function Programming (ICFP).*

[8] Bradford L Chamberlain, Steve Deitz, Mary Beth Hribar, and Wayne Wong. 2015. Chapel. *Programming Models for Parallel Computing* (2015), 129–159.

[9] The Scala Graph Project Contributors. 2023. Graph for Scala. https://www.scala-graph.org Version 2.0.1, Accessed: June 11, 2024.

[10] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. 2009. An Introduction to High-Level Synthesis. 26, 4 (2009), 8–17. https://doi.org/10.1109/MDT.2009.69

[11] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[12] S.A. Edwards. 2005. The challenges of hardware synthesis from C-like languages. In *Design, Automation and Test in Europe.* 66–67 Vol. 1. https://doi.org/10.1109/year.2005.307

[13] IEEE Floating Point Working Group. 2019. IEEE Standard for Floating-Point Arithmetic. (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[14] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff

Young, Xiang Zhou, Zongwei Zhou, and David Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. arXiv:2304.01433 [cs.AR] https://arxiv.org/abs/2304.01433

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society.

[16] MPI Consortium. 2009. *MPI: A Message-Passing Interface Standard Version 2.2*. Technical Report. Message Passing Interface Forum.

[17] Kir Nagaitsev, Karl Hallsby, Peizhi Liu, Lucas Myers, Alex Butler, David Krasowska, and Peter Dinda. 2025. Village: From High Level Parallelism to High Performance.

[18] NVIDIA Corporation. 2025. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Version 12.9.

[19] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. 2006. An Overview of the Scala Programming Language. https://www.scala-lang.org/docu/files/ScalaOverview.pdf

[20] Wilson Snyder, Paul Wasson, Duane Galbi, and et al. 2024. Verilator. https://github.com/verilator/verilator original-date: 2019-06-13T22:38:59Z.

[21] Victor Taelin. 2025. HVM2: A Parallel Evaluator for Interaction Combinators. https://raw.githubusercontent.com/HigherOrderCO/HVM/main/paper/HVM2.pdf

[22] ucb-bar. 2024. *Berkeley-Hardfloat*. UCB-BAR. https://github.com/ucb-bar/berkeley-hardfloat

[23] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanović. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).