Contributions to Aetherling

David Akeley

September 6, 2018

Abstract

This is a list of my contributions to the Aetherling project. I start with a section summarizing tasks I worked on, then I expand on the tasks in further sections.

1 Summary

1. Line Buffer Specifications

   I proposed a new specification of Aetherling's line buffer node and wrote a document ("The Line Buffer Manifesto") describing the benefits of this redesign. The previous line buffer design was hard to parallelize due to difficult-to-satisfy constraints on its parameters' divisibility, and also did not support downsampling ("stride"). The redesign addresses these issues.

2. Functional Simulator

   I wrote a functional simulator for Aetherling, which includes a simulation of the intended behavior of the redesigned line buffer. This allows the user to test the functionality of an Aetherling DAG using pure Haskell code.

3. Demonstration Apps

   I designed two demonstration Aetherling DAGs: Gaussian blur (7×7 stencil) and mipmap generation. These DAGs can be tested using the functional simulator along with a library (`ImageIO.hs`) for converting between simulator values and png images on disk. These apps also demonstrate the applications of the redesigned line buffer.

4. Simplifying Ops

   An op is a node of an Aetherling DAG (along with its children, sometimes). Previously there were multiple ways to express the same functionality. I

simplified the ops to minimize redundancy.

5. Helper Functions

   I wrote some Haskell helper functions for creating ops and simple patterns of ops. These helpers check for invalid parameters and substitute for functionality lost through the op simplification.

6. Ready-Valid Meta-Op

   By default, Aetherling pipelines are composed of ops representing circuits with synchronous timing: they wait for a certain number of warm-up cycles, then emit outputs on a repeating schedule (phase). I designed a `ReadyValid` op that represents the idea of wrapping a portion of an Aetherling DAG with a ready-valid interface. I modified the compose operators (`|&|` and `|>>=|`) to properly handle the 'ReadyValid' op. The user is prevented from composing an op with ready-valid timing with one with synchronous timing, and, when composing two ready-valid ops in sequence, the throughput-matching behavior of the Aetherling type system is suppressed.

7. ComposePar Retiming

   Aetherling includes a `ComposePar` op that represents placing circuits (child ops) in parallel. The user is not required to ensure that each parallel path has the same latency (sequential latency – the count of the number of register delays along a path). I wrote a pass that walks an Aetherling DAG searching for `ComposePar` ops, modifying its child ops if needed such that all paths have the same latency. The pass finds an optimal solution that minimizes the number of register bits added to the circuit.

8. Fractional Underutilization and Phase

   In the Aetherling team, "phase" refers to the repeating pattern of valid and garbage values input to/generated by a synchronously timed op. For example, an op that generated 1 valid output every 3 cycles would have an output phase of `[True, False, False]` (Two out of every three cycles, the op generates garbage).

   The choice of phase for an op with integer underutilization (1 valid input/output per X clock cycles) is obvious, but with fractional underutilization (X valid per Y clocks), there are several reasonable phase choices. When several such underutilized ops are joined together, there's no reason to assume that their phase patterns will match.

   Since the Aetherling type system only exposes type and throughput information to the user, it's vital that the system take care of phase matching

automatically. I proposed a scheme that assigns each fractional utilization ratio a standardized phase. (The earlier phase corresponds to $\frac{1}{3}$ utilization). This allows the complexity of phase matching to be confined to one op in the system, `SequenceArrayRepack`. Along with the ComposePar retiming pass, this makes it so that users only have to be concerned with type and throughput matching, allowing them to view phase and latency as performance, rather than correctness, issues.

9. Tests Written

   I gained a lot of experience writing tests as part of my work on the Aetherling project. These tests include tests for the functional simulator, tests for the hardware line buffer David Durst is designing, tests for the compose operators (`|&|` and `|>>=|`), and tests for the `ComposePar` retiming passes.

2 Line Buffer Specifications

3 Functional Simulator

4 Demonstration Apps

5 Simplifying Ops & Helper Functions

Previously, many Aetherling arithmetic ops (e.g. `Add`) took a type parameter, which could be an array type. This meant that there were multiple ways to express the same operation. For example, a bit xor could be expressed as `XOr T_Bit` or `Add T_Bit`, and elementwise addition of two 4-arrays-of-int could be expressed as `Add $ T_Array 4 T_Int` (4-array type parameter) or as a `MapOp 4` over a scalar `Add T_Int`.

Since each op of the Aetherling Haskell IR will eventually need to be implemented in hardware, it would be best to minimize unneeded functionality as much as possible. I eliminated the type parameter from arithmetic ops, splitting the op into two ops if bit and integer versions are both needed (e.g. 'And' for boolean *and*; 'AndInts' for bitwise *and*).

Since this change makes vectorized arithmetic harder to express directly, I wrote some helper functions for expressing array types and array operations. For example, a 16×16 matrix of integers can be expressed as `tInts[16,16]`, and an op performing elementwise addition of two matrices can be expressed as `addInts $`

`tInts[16,16]`. Internally, the IR represents vectorized ops as scalar ops wrapped by `MapOp`. In this example, `addInts` will return

```
MapOp 16 (MapOp 16 Add)
```

The helper functions also serve the purpose of checking for invalid parameters. For example, the function for creating a line buffer op[1] checks that the divisibility requirements are satisfied, and the `arrayReshape` function, which creates an `ArrayReshape` op that reinterprets inputs as a different type, checks that there's a reasonable mapping between input and output types.[2]

6 ReadyValid Op

7 ComposePar Retiming

8 Fractional Underutilization and Phase

9 Test Writing

The tests I designed include

1. Haskell tests for the functional simulator I designed.
2. Python tests for the hardware line buffer David Durst is implementing in Magma based on "The Line Buffer Manifesto" specifications. These tests use the CoreIR simulator.
3. Tests for the sequential compose (`|>>=|`) and parallel compose (`|&|`) Haskell Aetherling operators. These tests check that the produced Haskell IR node is correct given valid operands, and check that the operators reject invalid operands (mismatched port types, mismatched synchronous and ready-valid timing, and mismatched throughputs with synchronous timing).
4. Tests for the `ComposePar` retiming passes. There are 18+ test cases, some of which have compose ops nested several layers deep meant to check for corner cases.

[1] temporarily named `manifestoLineBuffer`

[2] Examples: `arrayReshape [tBits[2]] [T_Bit, T_Bit]` (conversion of 2-array-of-bit to two separate bits) is valid, while `arrayReshape [tBits[3]] [T_Bit]` is not since the input has more bits than the output.