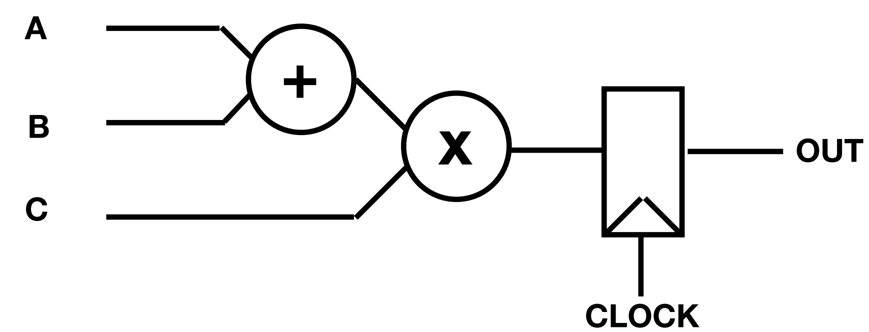
Synthesizable Higher-Order Functions for C++

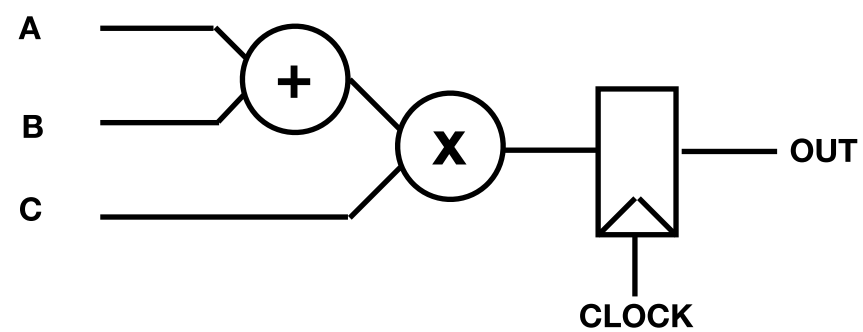
Dustin Richmond, Alric Althoff, Ryan Kastner

UC San Diego

Description vs Synthesis



Description vs Synthesis



Hardware Description Languages

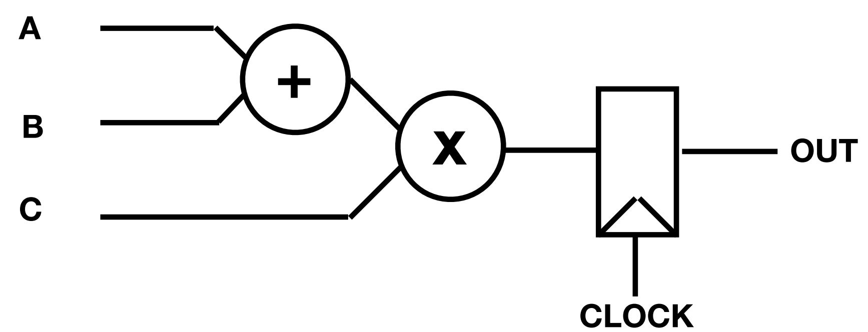
```
module demo
  (input           CLOCK,
   input [31:0]  A,
   input [31:0]  B,
   input [31:0]  C,
   output reg [31:0]  OUT);

   logic [31:0]          wResult;

   assign wResult = (A + B)*C;

   always @(posedge CLOCK) begin
      OUT <= wResult;
   end
endmodule
```

Description vs Synthesis



Hardware Description Languages

```
module demo
  (input           CLOCK,
   input [31:0]  A,
   input [31:0]  B,
   input [31:0]  C,
   output reg [31:0]  OUT);

   logic [31:0]          wResult;

   assign wResult = (A + B)*C;

   always @(posedge CLOCK) begin
      OUT <= wResult;
   end
endmodule
```

C++ Synthesis Languages

```
unsigned int demo(unsigned int A,
                  unsigned int B,
                  unsigned char C){

  return (A + B)  * C
}
```

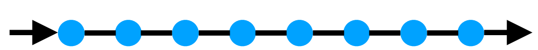
Parallel Patterns in Hardware

Genomics Databases Computer Vision Signals Sorting

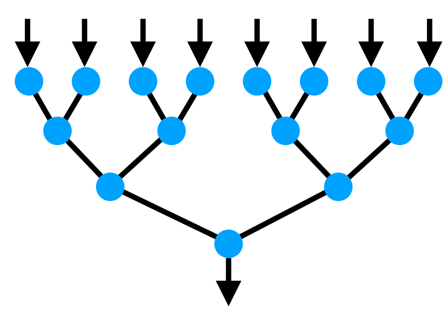
[1] J. Matai, **D. Richmond,** et al. "Resolve: Generation of high-performance sorting architectures from high-level synthesis," *ISFPGA, 2016*.
[2] D. Lee, **D. Richmond,** et al. "A streaming clustering approach using a heterogeneous system for big data analysis," *ICCAD 2017*.
[3] Q. Gautier, Quentin, A. Shearer, J. Matai, **D. Richmond**, et al. " Real-time 3D reconstruction for FPGAs," *FPT 2014.*
[4] **D. Richmond**, R. Kastner, A. Irturk and J. McGarry, "A FPGA design for high speed feature extraction from a compressed measurement stream," *FPL 2013.*
[5] E. Broussard, **D. Richmond**, et al. "A Model for Programming Data-Intensive Applications on FPGAs: A Genomics Case Study," *SAAHPC 2012.*

Parallel Patterns in Hardware

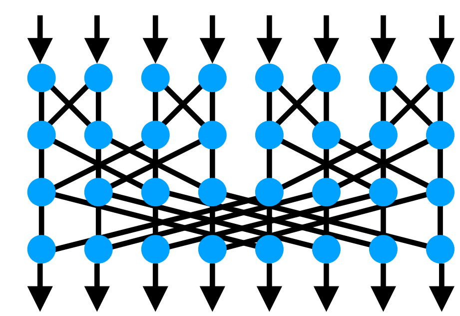
Genomics Databases Computer Vision Signals Sorting



Systolic Array



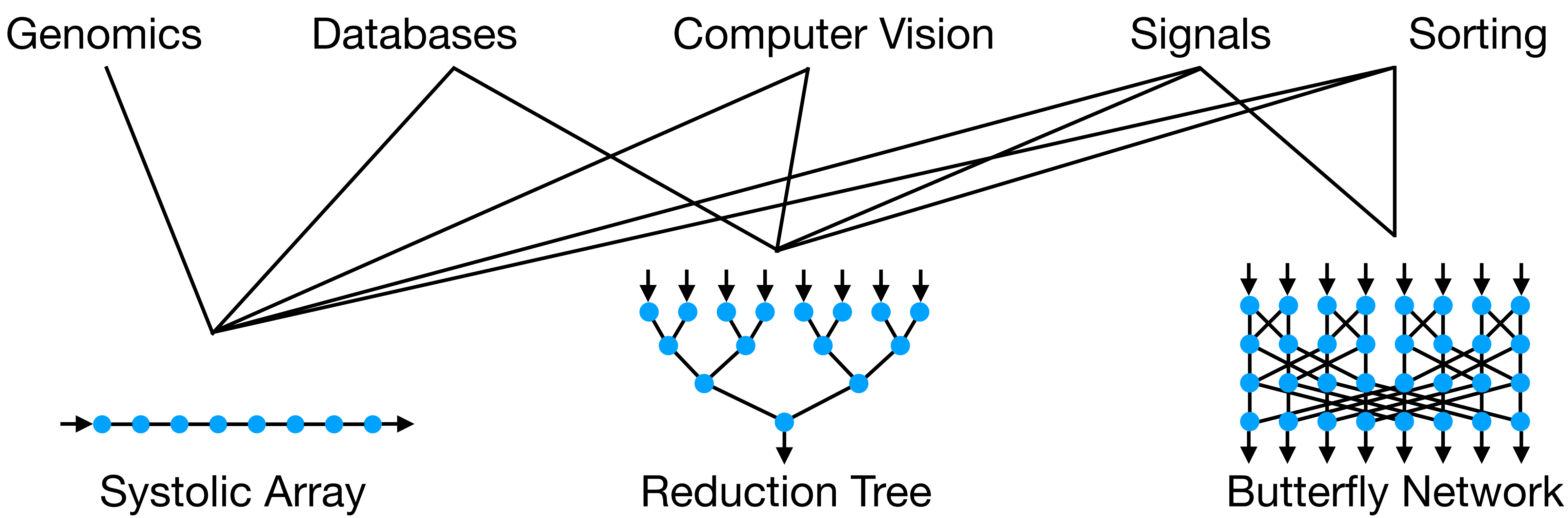
Reduction Tree



Butterfly Network

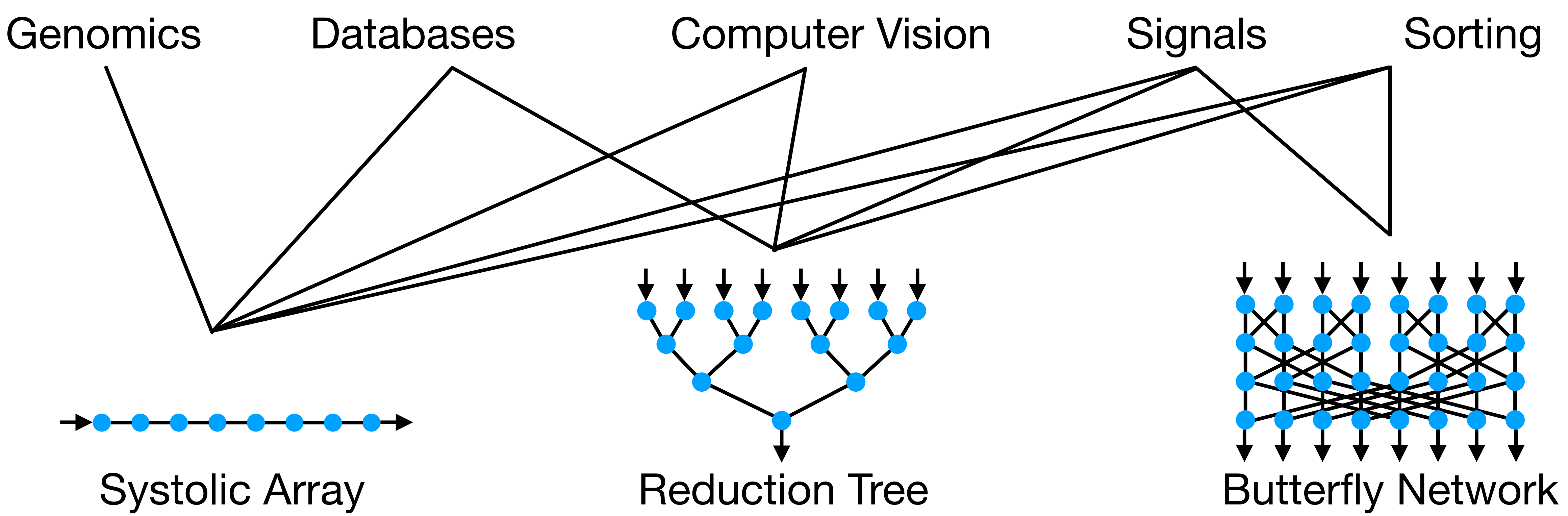
[1] J. Matai, **D. Richmond,** et al. "Resolve: Generation of high-performance sorting architectures from high-level synthesis," *ISFPGA, 2016*.
[2] D. Lee, **D. Richmond,** et al. "A streaming clustering approach using a heterogeneous system for big data analysis," *ICCAD 2017*.
[3] Q. Gautier, Quentin, A. Shearer, J. Matai, **D. Richmond**, et al. " Real-time 3D reconstruction for FPGAs," *FPT 2014.*
[4] **D. Richmond**, R. Kastner, A. Irturk and J. McGarry, "A FPGA design for high speed feature extraction from a compressed measurement stream," *FPL 2013.*
[5] E. Broussard, **D. Richmond**, et al. "A Model for Programming Data-Intensive Applications on FPGAs: A Genomics Case Study," *SAAHPC 2012.*

Parallel Patterns in Hardware



[1] J. Matai, **D. Richmond,** et al. "Resolve: Generation of high-performance sorting architectures from high-level synthesis," *ISFPGA, 2016*.
[2] D. Lee, **D. Richmond,** et al. "A streaming clustering approach using a heterogeneous system for big data analysis," *ICCAD 2017*.
[3] Q. Gautier, Quentin, A. Shearer, J. Matai, **D. Richmond**, et al. " Real-time 3D reconstruction for FPGAs," *FPT 2014.*
[4] **D. Richmond**, R. Kastner, A. Irturk and J. McGarry, "A FPGA design for high speed feature extraction from a compressed measurement stream," *FPL 2013.*
[5] E. Broussard, **D. Richmond**, et al. "A Model for Programming Data-Intensive Applications on FPGAs: A Genomics Case Study," *SAAHPC 2012.*

Parallel Patterns in Hardware



Many Applications -> Small Number of Patterns

[1] J. Matai, **D. Richmond,** et al. "Resolve: Generation of high-performance sorting architectures from high-level synthesis," *ISFPGA, 2016*.
[2] D. Lee, **D. Richmond,** et al. "A streaming clustering approach using a heterogeneous system for big data analysis," *ICCAD 2017*.
[3] Q. Gautier, Quentin, A. Shearer, J. Matai, **D. Richmond**, et al. " Real-time 3D reconstruction for FPGAs," *FPT 2014.*
[4] **D. Richmond**, R. Kastner, A. Irturk and J. McGarry, "A FPGA design for high speed feature extraction from a compressed measurement stream," *FPL 2013.*
[5] E. Broussard, **D. Richmond**, et al. "A Model for Programming Data-Intensive Applications on FPGAs: A Genomics Case Study," *SAAHPC 2012.*

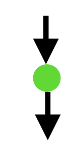
Higher-Order Functions as Patterns

Higher-Order Functions as Patterns

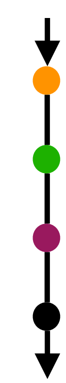
```
i = 10.0
r = apply(square, i)
# r = 100.0
```

Higher-Order Functions as Patterns

```
i = 10.0
r = apply(square, i)
# r = 100.0
```

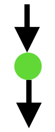


```
a = 1
st1 = compose(square, add1)
st2 = compose(mulby2, st1)
st3 = compose(divby4, st3)
b = st2(a)
# b = 2 ((((a + 1) ^ 2) * 2) / 4)
```

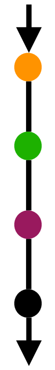


Higher-Order Functions as Patterns

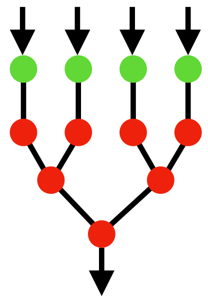
```
i = 10.0
r = apply(square, i)
# r = 100.0
```



```
a = 1
st1 = compose(square, add1)
st2 = compose(mulby2, st1)
st3 = compose(divby4, st3)
b = st2(a)
# b = 2 ((((a + 1) ^ 2) * 2) / 4)
```



```
l = [1, 2, 3, 4]
m = map(square, l)
r = reduce(sum, m)
# r = 30
```



Pointers and Polymorphism

- Lists
- Recursion (Loops)
- Function-Passsing (First Class Functions)

Research Questions

- Can we create synthesizable Higher-Order Functions?[6, 7]
- How does the syntax compare to Python?
- Is there a performance, area, or frequency cost?

[6] L. Josipovic, et al., "Enriching C-based High-Level Synthesis with parallel pattern templates," *ICFPT 2016*.
[7] R. Prabhakar, et al., "Generating configurable hardware from parallel patterns," *ACM SIGARCH 2016*.

Function Passing

```
int square(int &input){
  return input*input;
}

int apply(int (*f)(int&), int &input){
  return f(input);
}

int main(){
  int res;
  res = apply(square, 10);
  // res = 100
  return 0;
}
```

Function Passing

```
int square(int &input){
  return input*input;
}

int apply(int (*f)(int&), int &input){
  return f(input);
}

int main(){
  int res;
  res = apply(square, 10);
  // res = 100
  return 0;
}
```

Pointer! (Not *Synthesizable*)

Function Passing

```
struct Square{
  int operator()(int &input){
    return input*input;
  }
};

int apply(Square &f, int &input){
  return f(input);
}

int main(){
  int res = apply(Square(), 10);
  // res = 100
  return 0;
}
```

Function Passing

```
struct Square{
  int operator()(int &input){
    return input*input;
  }
};

int apply(Square &f, int &input){
  return f(input);
}

int main(){
  int res = apply(Square(), 10);
  // res = 100
  return 0;
}
```

Object Reference (Synthesizable)

Polymorphism

```
struct Square{
  template <typename TI>
  TI operator()(TI &input){
    return input*input;
  }
};

template <typename FN, typename TI>
auto apply(FN &f, TI &input){
  return f(input);
}

int main(){
  int res = apply(Square(), 10);
  // res = 100
  return 0;
}
```

Polymorphism

```
struct Square{
  template <typename TI>
  TI operator()(TI &input){
    return input*input;
  }
};

template <typename FN, typename TI>
auto apply(FN &f, TI &input){
  return f(input);
}

int main(){
  int res = apply(Square(), 10);
  // res = 100
  return 0;
}
```

Type Templates

Polymorphism

Type Templates

```
struct Square{
  template <typename TI>
  TI operator()(TI &input){
    return input*input;
  }
};

template <typename FN, typename TI>
auto apply(FN &f, TI &input){
  return f(input);
}

int main(){
  int res = apply(Square(), 10);
  // res = 100
  return 0;
}
```

Return-Type Inference

Polymorphism

```
struct Square{
  template <typename TI>
  TI operator()(TI &input){
    return input*input;
  }
};

template <typename FN, typename TI>
auto apply(FN &f, TI &input){
  return f(input);
}

int main(){
  int res = apply(Square(), 10);
  // res = 100
  return 0;
}
```

Type Templates

Return-Type Inference

Template Inference

Interested?

Come visit my poster!

Implementation: `reduce`

Implementation: `reduce`

```
template <size_t LEN>
struct ReduceHelper{
    template<typename FN, typename TI, typename TA>
    auto operator()(FN &F, TI INIT, array<TA, LEN> IN){
        return ReduceHelper<FN, LEN-1>()(
            F(INIT, head(IN)), tail(IN));
    }
};

template <>
struct ReduceHelper<0>{
    template<typename FN, typename TI, typename TA>
    TI operator()(FN &F, TI INIT, array<TA, 0> IN){
        return INIT;
    }
};

template <typename FN, typename TI, typename TA, size_t LEN>
auto reduce(FN &F, TI INIT, array<TA, LEN> IN){
    return ReduceHelper<LEN>()(F, INIT, IN);
}
```

Implementation: `reduce`

```
template <size_t LEN>
struct ReduceHelper{
    template<typename FN, typename TI, typename TA>
    auto operator()(FN &F, TI INIT, array<TA, LEN> IN){
        return ReduceHelper<FN, LEN-1>()(
            F(INIT, head(IN)), tail(IN));
    }
};

template <>
struct ReduceHelper<0>{
    template<typename FN, typename TI, typename TA>
    TI operator()(FN &F, TI INIT, array<TA, 0> IN){
        return INIT;
    }
};

template <typename FN, typename TI, typename TA, size_t LEN>
auto reduce(FN &F, TI INIT, array<TA, LEN> IN){
    return ReduceHelper<LEN>()(F, INIT, IN);
}
```

Type Templates

Implementation: `reduce`

Return-Type Inference

Type Templates

```
template <size_t LEN>
struct ReduceHelper{
    template<typename FN, typename TI, typename TA>
    auto operator()(FN &F, TI INIT, array<TA, LEN> IN){
        return ReduceHelper<FN, LEN-1>()(
            F(INIT, head(IN)), tail(IN));
    }
};

template <>
struct ReduceHelper<0>{
    template<typename FN, typename TI, typename TA>
    TI operator()(FN &F, TI INIT, array<TA, 0> IN){
        return INIT;
    }
};

template <typename FN, typename TI, typename TA, size_t LEN>
auto reduce(FN &F, TI INIT, array<TA, LEN> IN){
    return ReduceHelper<LEN>()(F, INIT, IN);
}
```

Implementation: `reduce`

```
template <size_t LEN>
struct ReduceHelper{
    template<typename FN, typename TI, typename TA>
    auto operator()(FN &F, TI INIT, array<TA, LEN> IN){
        return ReduceHelper<FN, LEN-1>()(
            F(INIT, head(IN)), tail(IN));
    }
};

template <>
struct ReduceHelper<0>{
    template<typename FN, typename TI, typename TA>
    TI operator()(FN &F, TI INIT, array<TA, 0> IN){
        return INIT;
    }
};

template <typename FN, typename TI, typename TA, size_t LEN>
auto reduce(FN &F, TI INIT, array<TA, LEN> IN){
    return ReduceHelper<LEN>()(F, INIT, IN);
}
```

Return-Type Inference

Type Templates

Object Reference

Implementation: `reduce`

Static Recursion

Return-Type Inference

Type Templates

```
template <size_t LEN>
struct ReduceHelper{
    template<typename FN, typename TI, typename TA>
    auto operator()(FN &F, TI INIT, array<TA, LEN> IN){
        return ReduceHelper<FN, LEN-1>()(
            F(INIT, head(IN)), tail(IN));
    }
};

template <>
struct ReduceHelper<0>{
    template<typename FN, typename TI, typename TA>
    TI operator()(FN &F, TI INIT, array<TA, 0> IN){
        return INIT;
    }
};

template <typename FN, typename TI, typename TA, size_t LEN>
auto reduce(FN &F, TI INIT, array<TA, LEN> IN){
    return ReduceHelper<LEN>()(F, INIT, IN);
}
```

Object Reference

Implementation: `reduce`

Static Recursion

Return-Type Inference

Type Templates

```
template <size_t LEN>
struct ReduceHelper{
    template<typename FN, typename TI, typename TA>
    auto operator()(FN &F, TI INIT, array<TA, LEN> IN){
        return ReduceHelper<FN, LEN-1>()(
            F(INIT, head(IN)), tail(IN));
    }
};

template <>
struct ReduceHelper<0>{
    template<typename FN, typename TI, typename TA>
    TI operator()(FN &F, TI INIT, array<TA, 0> IN){
        return INIT;
    }
};

template <typename FN, typename TI, typename TA, size_t LEN>
auto reduce(FN &F, TI INIT, array<TA, LEN> IN){
    return ReduceHelper<LEN>()(F, INIT, IN);
}
```

Arrays/Lists

Object Reference

Complexity Begets Simplicity

```
array<int, 4> l = {1, 2, 3, 4};
r = reduce(add, l, 0);
// r = 10
```

Complexity Begets Simplicity

```
array<int, 4> l = {1, 2, 3, 4};
r = reduce(std::plus<int>(), l, 0);
// r = 10
```

Research Questions

- Can we create synthesizable Higher-Order Functions?[6, 7]
- How does the syntax compare to Python?
- Is there a performance, area, or frequency cost?

[6] L. Josipovic, et al., "Enriching C-based High-Level Synthesis with parallel pattern templates," *ICFPT 2016*.
[7] R. Prabhakar, et al., "Generating configurable hardware from parallel patterns," *ACM SIGARCH 2016*.

Syntax Comparison

Python	**C++**

Syntax Comparison

	Python	C++

Functions	`def add(l, r):`
	`    return (l + r)`
		`struct Add{`
		`  int operator()(int l, int r){`
		`    return (l + r);`
		`  }`
		`} add;`

Syntax Comparison

	Python	C++

Functions	`def add(l, r):`
	`    return (l + r)`
		`struct Add{`
		`  int operator()(int l, int r){`
		`    return (l + r);`
		`  }`
		`} add;`

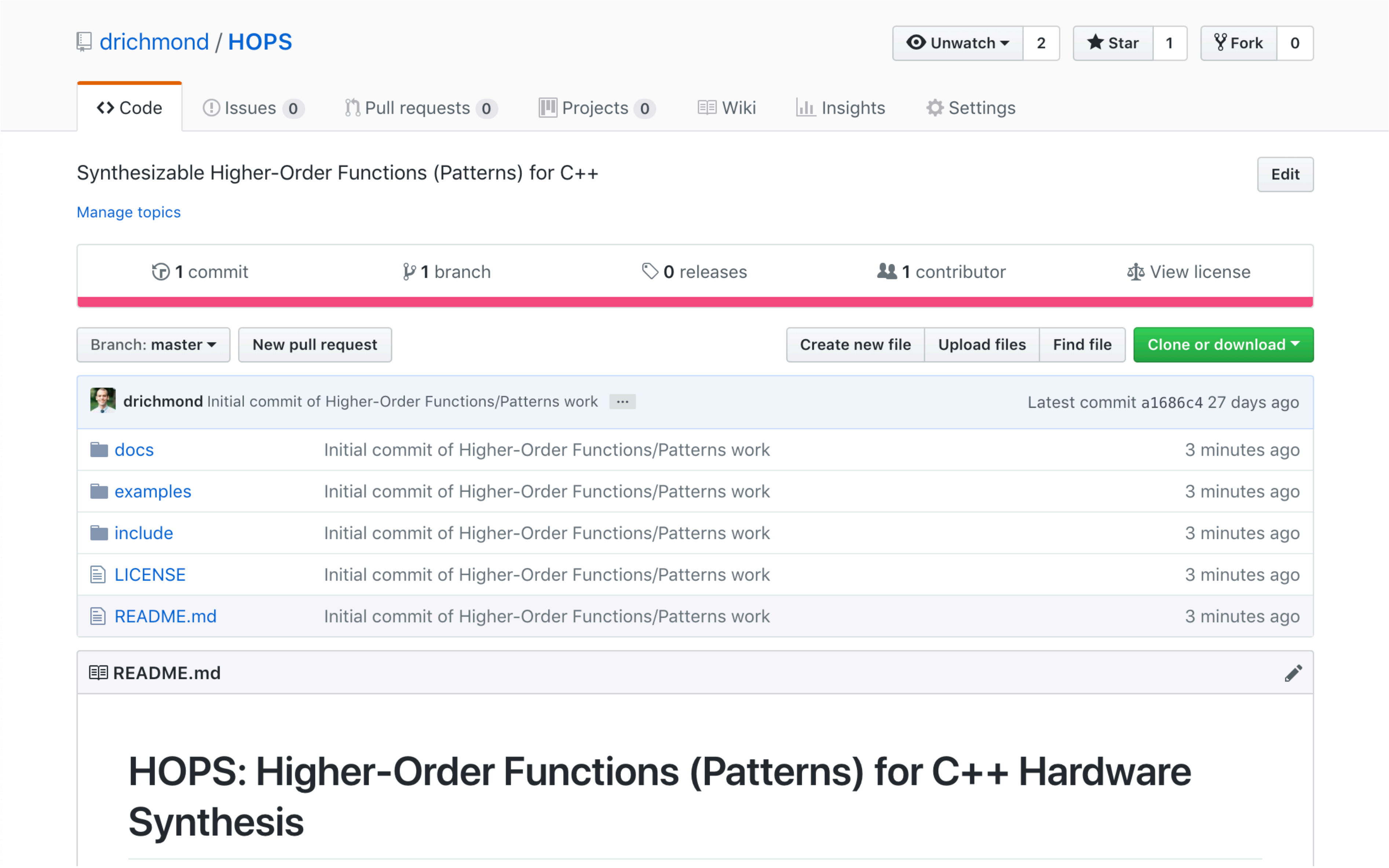
Map-Reduce	`l = [1, 2, 3, 4]`
	`m = map(square, l)`
	`r = reduce(add, m)`
		`array<int, 4> l = {1, 2, 3, 4};`
		`array<int, 4> m, r;`
		`m = map(square, l);`
		`r = reduce(add, m);`

Syntax Differences

- Wrapped Functions (a.k.a “Functors”)
- Explicit Typing
- No Tuples*

Clone our work!

github.com/drichmond/hops



Research Questions

- Can we create synthesizable Higher-Order Functions?[6, 7]
- How does the syntax compare to Python?
- Is there a performance, area, or frequency cost?

[6] L. Josipovic, et al., "Enriching C-based High-Level Synthesis with parallel pattern templates," *ICFPT 2016*.
[7] R. Prabhakar, et al., "Generating configurable hardware from parallel patterns," *ACM SIGARCH 2016*.

Cost Analysis

- 6 Application Functions
- 13 Maximum Frequency Datapoints
- Null Hypothesis: Mean max frequencies are not equal

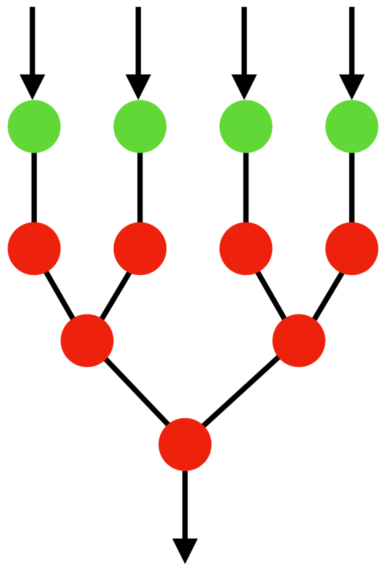
[1] J. Matai, **D. Richmond,** et al. "Resolve: Generation of high-performance sorting architectures from high-level synthesis," *ISFPGA, 2016*.
[2] D. Lee, **D. Richmond,** et al. "A streaming clustering approach using a heterogeneous system for big data analysis," *ICCAD 2017*.
[3] Q. Gautier, Quentin, A. Shearer, J. Matai, **D. Richmond**, et al. " Real-time 3D reconstruction for FPGAs," *FPT 2014.*
[4] **D. Richmond**, R. Kastner, A. Irturk and J. McGarry, "A FPGA design for high speed feature extraction from a compressed measurement stream," *FPL 2013.*
[5] E. Broussard, **D. Richmond**, et al. "A Model for Programming Data-Intensive Applications on FPGAs: A Genomics Case Study," *SAAHPC 2012.*

Experimental Setup

“Traditional”
(Trad.)

```
for(/*...*/){
  // Map Implementation
}

for(/*...*/){
  // Reduce Implementation
}
```



Higher-Order Functions
(H.O.F.)

```
m = map(/*Map Fn*/, l);
r = reduce(/*Reduce Fn*/, m);
```

Performance: Throughput & Latency

	Throughput (Results/Cycle)		Latency (Cycles)	
	H.O.F	Traditional	H.O.F.	Traditional
Finite Impulse Response Filter	1	1	65	65
Insertion Sort	1	1	31	31
Smith Waterman	1	1	16	16
ArgMin	1	1	7	7
Fast Fourier Transform	1	1	59	59
Bitonic Sort	1	1	21	21

Area: Resources Consumed

	Flip-Flops		SRL		LUT		DSPs	
	H.O.F.	Trad.	H.O.F.	Trad.	H.O.F.	Trad.	H.O.F.	Trad.
Finite Impulse Response Filter	14388	14388	**227**	**272**	**7306**	**7305**	48	48
Insertion Sort	2300	2300	0	0	935	935	0	0
Smith Waterman	895	895	11	11	**1187**	**1186**	0	0
ArgMin	**2670**	**2666**	**8**	**10**	**1575**	**1573**	0	0
Fast Fourier Transform	**21263**	**21240**	**2487**	**2494**	8096	8096	77	77
Bitonic Sort	11929	11929	1	1	4869	4869	0	0

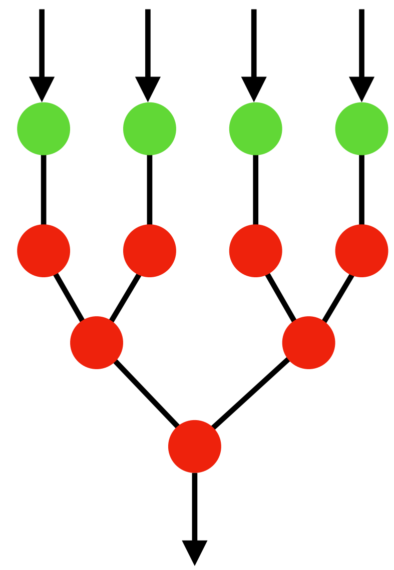
Maximum Frequency

	Higher-Order Function		Traditional		$\alpha < .05$
	Mean (μ, MHz)	Std Dev (MHz)	Mean (μ, MHz)	Std. Dev (MHz)	$\mu_{Loop} \neq \mu_{H.O.F.}$
Finite Impulse Response Filter	166.80	3.12	165.56	3.57	0.01
Insertion Sort	162.83	3.22	166.47	5.02	0.008
Smith Waterman	103.73	4.19	103.58	5.34	0.025
ArgMin	110.64	2.51	110.91	2.89	0.02
Fast Fourier Transform	123.56	3.35	123.56	3.18	**0.05**
Bitonic Sort	112.77	3.50	113.85	2.76	0.01

Conclusion:

No performance, area, or frequency cost

Synthesizable Higher-Order Functions for C++



- C++ Higher-Order Functions use Meta-Programming
- Provide a Python-like syntax
- No performance, area, or frequency cost

github.com/drichmond/hops