BitSAD v2: Compiler Optimization and Analysis for Bitstream Computing

HiPEAC 2020 (published in TACO)

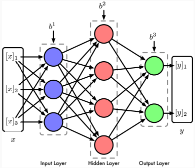
Kyle Daruwalla, Heng Zhuo, Rohit Shukla, and Mikko Lipasti

20 January 2020

University of Wisconsin - Madison, Dept. of Elec. and Comp. Eng.

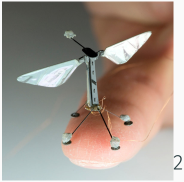
Motivation

CV/ML algorithms enable powerful new applications




[1]

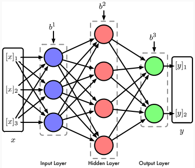
Fabrication techniques enabling pico-aerial vehicles (PAVs)


[2]

[1] Dllu 2017

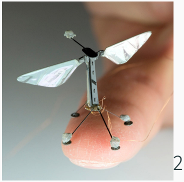
[2] Ma 2015

CV/ML algorithms enable powerful new applications



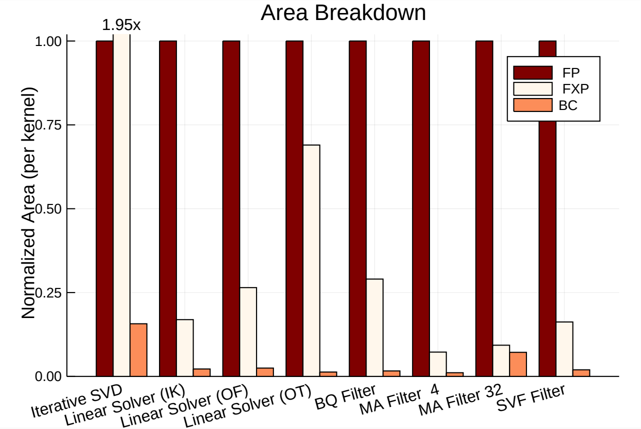

[1]

Fabrication techniques enabling pico-aerial vehicles (PAVs)


[2]

How do we enable advanced CV/ML algorithms on ultra-low power platforms?

Motivation



The resource consumption of bitstream computing (BC) implementations is much lower than floating point (FP) and fixed point (FXP) designs.

Overview

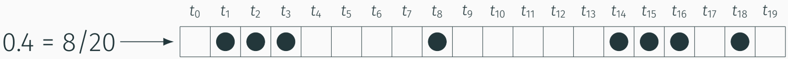
1. Background: Bitstreams and example applications
2. BitSAD: What is it good for? [3]
3. Population coding: Parallelization for stochastic computing
4. Optimizations
   - 4.1 for stochastic bitstreams
   - 4.2 for deterministic bitstreams

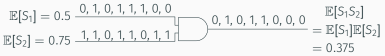
[3] Seinfeld: S5E14, Feb. 1994.

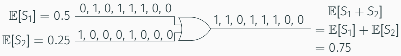
Background: Bitstream Computing

What is Bitstream Computing?

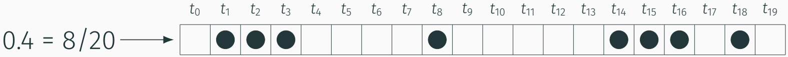
Stochastic Bitstreams:

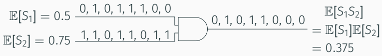


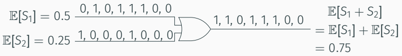




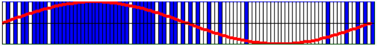
Stochastic Bitstreams:







Deterministic Bitstreams:

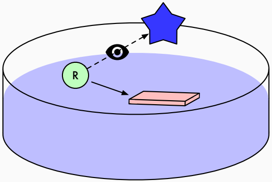


Density of "1" ⇒ Higher amplitude

- Sequence is *deterministic*
- Oversampled audio data
- Leads to efficient filters

Example Application #1: Navigation

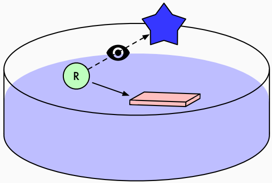
Consider a robot (denoted “R”) that needs to navigate an unknown environment

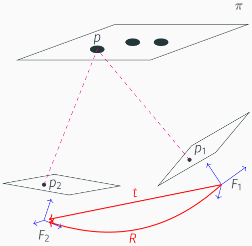


[4] Malis and Vargas 2007.

Example Application #1: Navigation

Consider a robot (denoted “R”) that needs to navigate an unknown environment

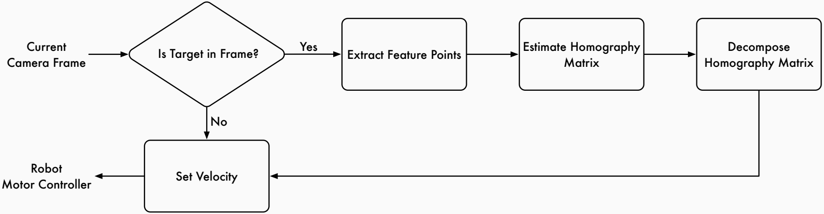




Navigation by visual cues — homography estimation and decomposition[4]

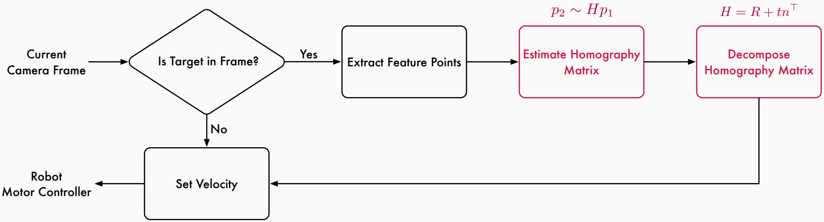
[4] Malis and Vargas 2007.

Example Application #1: Navigation Pipeline



[5] Dubrofsky 2009.

Example Application #1: Navigation Pipeline



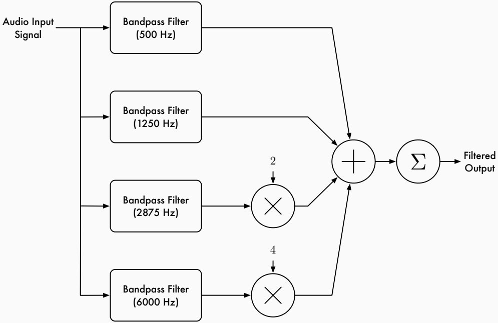
Requires linear solver and singular value decomposition[5]

[5] Dubrofsky 2009.

Example Application #2: Channel Shaping

Audio information is sent across several frequency channels

Need to shape input signal to focus on channels of interest

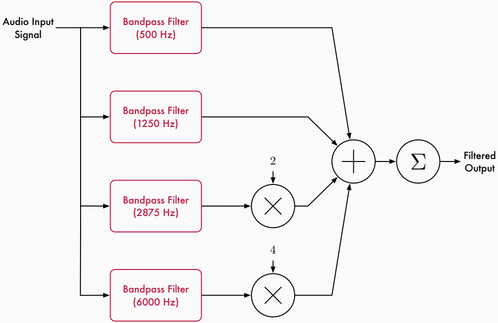


Example Application #2: Channel Shaping

Audio information is sent across several frequency channels

Need to shape input signal to focus on channels of interest

Oversampled bitstreams ⇒ more efficient filters



BitSAD

What is BitSAD?

A domain-specific language for bitstream computing

Allows users to simulate bit-level, cycle-accurate designs

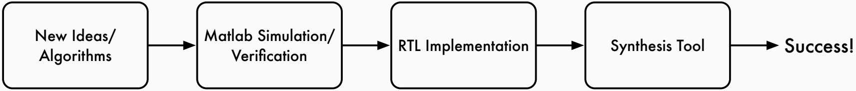
What is BitSAD?

A domain-specific language for bitstream computing

Allows users to simulate bit-level, cycle-accurate designs *and* automatically generate Verilog implementations!

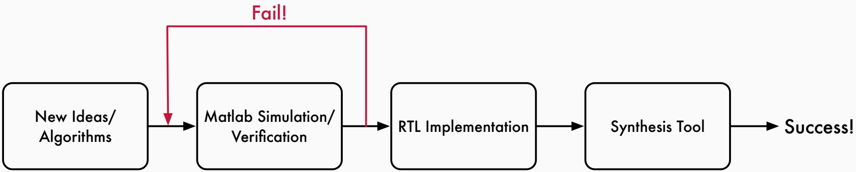
Typical Design Flow

Traditional design flow turnover is slow:



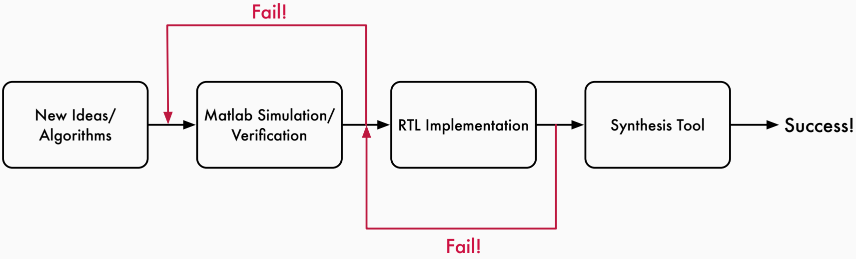
Typical Design Flow

Traditional design flow turnover is slow:



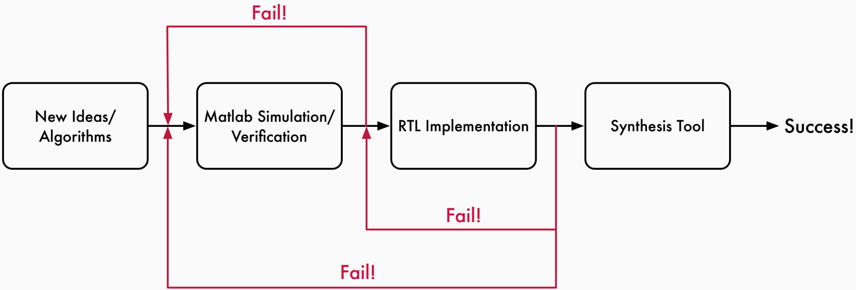
Typical Design Flow

Traditional design flow turnover is slow:



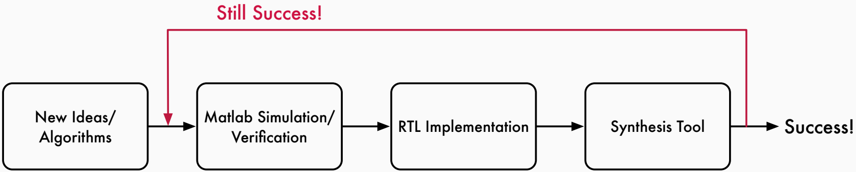
Typical Design Flow

Traditional design flow turnover is slow:



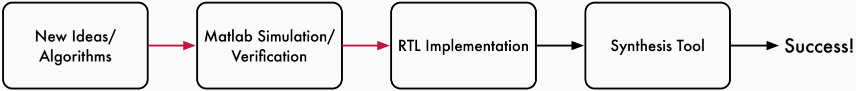
Typical Design Flow

Traditional design flow turnover is slow:



Typical Design Flow

Where the problem is coming from:



Typical Design Flow

What can we do about it:

BitSAD by Example

Algorithm 1 Iterative SVD

Require: Input matrix $A \in \mathbb{R}^{m \times n}$ and initial guess $v_0 \in \mathbb{R}^n$

for $k = 1, 2, \dots$ (until convergence) **do**

$\quad w_k = A v_{k-1}$

$\quad \alpha_k = \|w_k\|_2 = \sqrt{w_k^\top w_k}$

$\quad u_k = w_k / \alpha_k$

$\quad z_k = A^\top u_k$

$\quad \sigma_k = \|z_k\|_2 = \sqrt{z_k^\top z_k}$

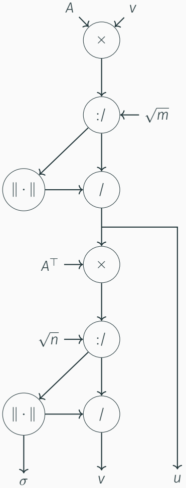
$\quad v_k = z_k / \sigma_k$

end for

return First left/right singular vectors, u_k & v_k, and first singular value, σ_k

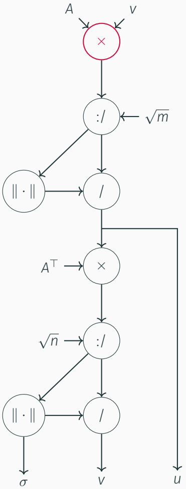
```
def loop(A: Matrix[SBitstream],
         v: Matrix[SBitstream]):
    (Matrix[SBitstream],
     Matrix[SBitstream], SBitstream) = {

  // Update right singular vector
  var w = A * v
  var wScaled = w :/ math.sqrt(params.m)
  var u = wScaled / Matrix.norm(wScaled)

  // Update left singular vector
  var z = A.T * u
  var zScaled = z :/ math.sqrt(params.n)
  var sigma = Matrix.norm(zScaled)
  var _v = zScaled / sigma

  (u, _v, sigma)
}
```

BitSAD by Example



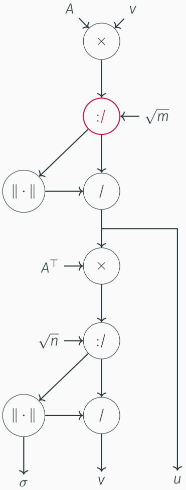

```
def loop(A: Matrix[SBitstream],
         v: Matrix[SBitstream]):
    (Matrix[SBitstream],
     Matrix[SBitstream], SBitstream) = {

  // Update right singular vector
  var w = A * v
  var wScaled = w :/ math.sqrt(params.m)
  var u = wScaled / Matrix.norm(wScaled)

  // Update left singular vector
  var z = A.T * u
  var zScaled = z :/ math.sqrt(params.n)
  var sigma = Matrix.norm(zScaled)
  var _v = zScaled / sigma

  (u, _v, sigma)
}
```

BitSAD by Example



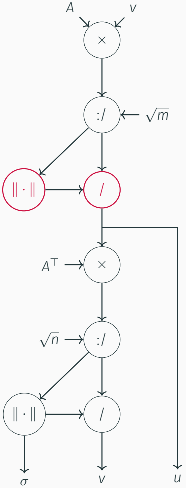

```
def loop(A: Matrix[SBitstream],
         v: Matrix[SBitstream]):
    (Matrix[SBitstream],
     Matrix[SBitstream], SBitstream) = {

  // Update right singular vector
  var w = A * v
  var wScaled = w :/ math.sqrt(params.m)
  var u = wScaled / Matrix.norm(wScaled)

  // Update left singular vector
  var z = A.T * u
  var zScaled = z :/ math.sqrt(params.n)
  var sigma = Matrix.norm(zScaled)
  var _v = zScaled / sigma

  (u, _v, sigma)
}
```

BitSAD by Example



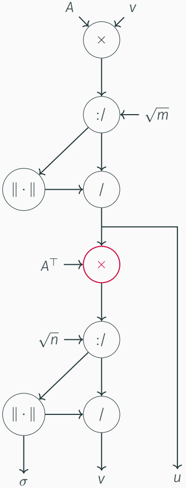

```
def loop(A: Matrix[SBitstream],
         v: Matrix[SBitstream]):
  (Matrix[SBitstream],
   Matrix[SBitstream], SBitstream) = {

  // Update right singular vector
  var w = A * v
  var wScaled = w :/ math.sqrt(params.m)
  var u = wScaled / Matrix.norm(wScaled)

  // Update left singular vector
  var z = A.T * u
  var zScaled = z :/ math.sqrt(params.n)
  var sigma = Matrix.norm(zScaled)
  var _v = zScaled / sigma

  (u, _v, sigma)
}
```

BitSAD by Example

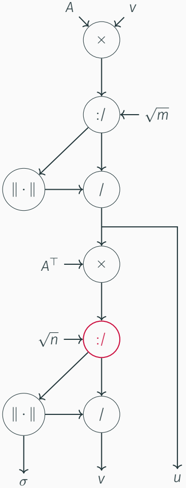



```
def loop(A: Matrix[SBitstream],
         v: Matrix[SBitstream]):
    (Matrix[SBitstream],
     Matrix[SBitstream], SBitstream) = {

  // Update right singular vector
  var w = A * v
  var wScaled = w :/ math.sqrt(params.m)
  var u = wScaled / Matrix.norm(wScaled)

  // Update left singular vector
  var z = A.T * u
  var zScaled = z :/ math.sqrt(params.n)
  var sigma = Matrix.norm(zScaled)
  var _v = zScaled / sigma

  (u, _v, sigma)
}
```

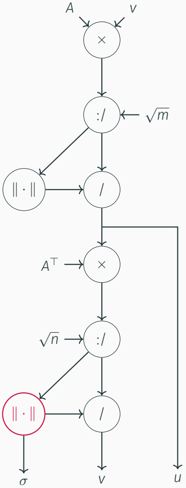
BitSAD by Example




```
def loop(A: Matrix[SBitstream],
         v: Matrix[SBitstream]):
    (Matrix[SBitstream],
     Matrix[SBitstream], SBitstream) = {

  // Update right singular vector
  var w = A * v
  var wScaled = w :/ math.sqrt(params.m)
  var u = wScaled / Matrix.norm(wScaled)

  // Update left singular vector
  var z = A.T * u
  var zScaled = z :/ math.sqrt(params.n)
  var sigma = Matrix.norm(zScaled)
  var _v = zScaled / sigma

  (u, _v, sigma)
}
```

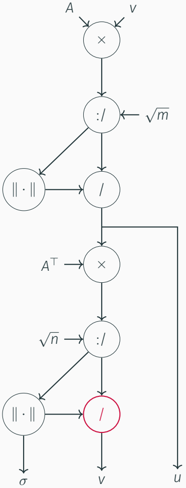



```
def loop(A: Matrix[SBitstream],
         v: Matrix[SBitstream]):
    (Matrix[SBitstream],
     Matrix[SBitstream], SBitstream) = {

  // Update right singular vector
  var w = A * v
  var wScaled = w :/ math.sqrt(params.m)
  var u = wScaled / Matrix.norm(wScaled)

  // Update left singular vector
  var z = A.T * u
  var zScaled = z :/ math.sqrt(params.n)
  var sigma = Matrix.norm(zScaled)
  var _v = zScaled / sigma

  (u, _v, sigma)
}
```



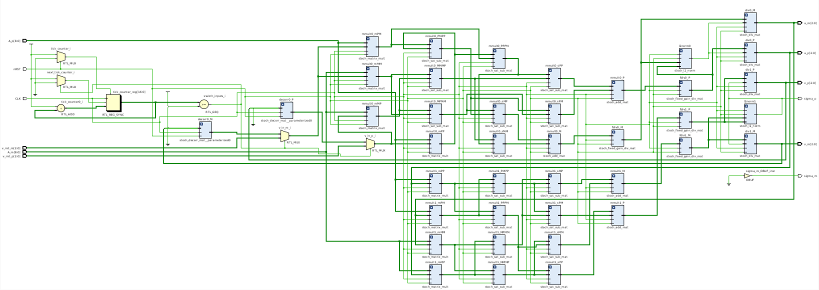

```
def loop(A: Matrix[SBitstream],
         v: Matrix[SBitstream]):
    (Matrix[SBitstream],
     Matrix[SBitstream], SBitstream) = {

  // Update right singular vector
  var w = A * v
  var wScaled = w :/ math.sqrt(params.m)
  var u = wScaled / Matrix.norm(wScaled)

  // Update left singular vector
  var z = A.T * u
  var zScaled = z :/ math.sqrt(params.n)
  var sigma = Matrix.norm(zScaled)
  var _v = zScaled / sigma

  (u, _v, sigma)
}
```

BitSAD by Example




```
def loop(A: Matrix[SBitstream],
         v: Matrix[SBitstream]):
    (Matrix[SBitstream],
     Matrix[SBitstream], SBitstream) = {

  // Update right singular vector
  var w = A * v
  var wScaled = w :/ math.sqrt(params.m)
  var u = wScaled / Matrix.norm(wScaled)

  // Update left singular vector
  var z = A.T * u
  var zScaled = z :/ math.sqrt(params.n)
  var sigma = Matrix.norm(zScaled)
  var _v = zScaled / sigma

  (u, _v, sigma)
}
```

BitSAD by Example



BitSAD Summary

- Allows designers to write algorithms at a high abstraction level
- Simulates designs with bit-level, cycle-accurate results
- Generates synthesizable hardware with no code changes

BitSAD Summary

- Allows designers to write algorithms at a high abstraction level
- Simulates designs with bit-level, cycle-accurate results
- Generates synthesizable hardware with no code changes

Is this enough to realize "general-purpose" bitstream computing?

Population Coding

Latency Issue for Stochastic Bitstreams

Recall the definition of a stochastic bitstream:

$$\frac{1}{T}\sum_{t=1}^{T} X_t \approx \mathbb{E}X_t = p \quad \text{as } T \to \infty$$

Latency Issue for Stochastic Bitstreams

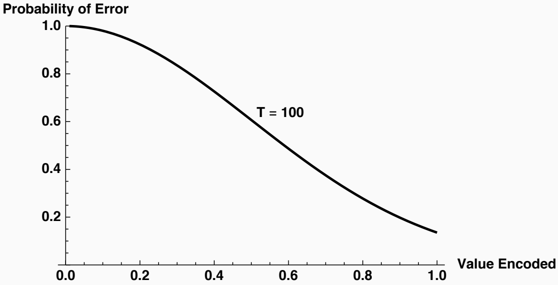
Recall the definition of a stochastic bitstream:

$$\frac{1}{T}\sum_{t=1}^{T} X_t \approx \mathbb{E}X_t = p \quad \text{as } T \to \infty$$

How large should T be for 10% application error? 5% error?

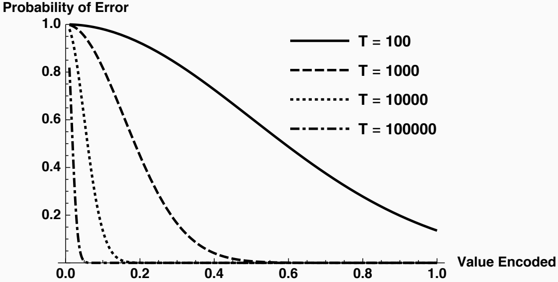
Latency Issue for Stochastic Bitstreams

Quantify error using Hoeffding's inequality:



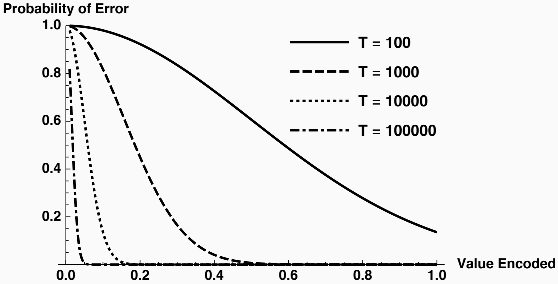
Latency Issue for Stochastic Bitstreams

Quantify error using Hoeffding's inequality:



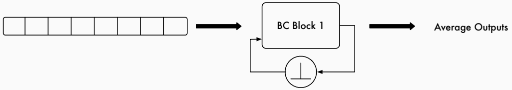
Latency Issue for Stochastic Bitstreams

Quantify error using Hoeffding's inequality:



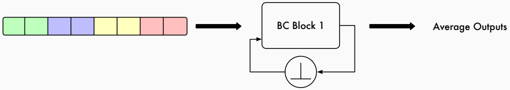
Error depends on # of time steps, relative accuracy, *and* magnitude of number!

Parallelizing Stochastic Bitstreams



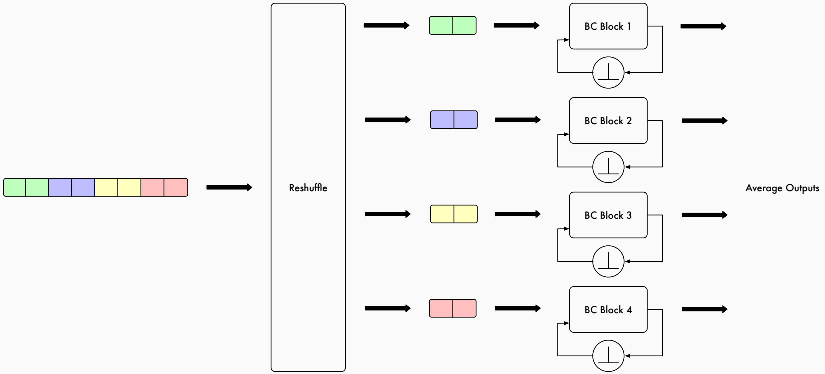
$$p \approx \frac{1}{T} \sum_{t=1}^{T} X_t$$

Parallelizing Stochastic Bitstreams



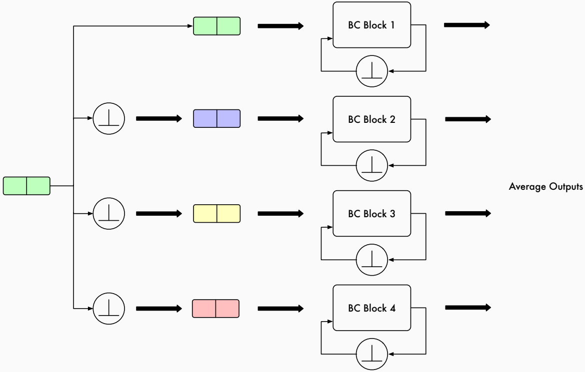
$$p \approx \frac{1}{T} \sum_{t=1}^{T} X_t$$

Parallelizing Stochastic Bitstreams



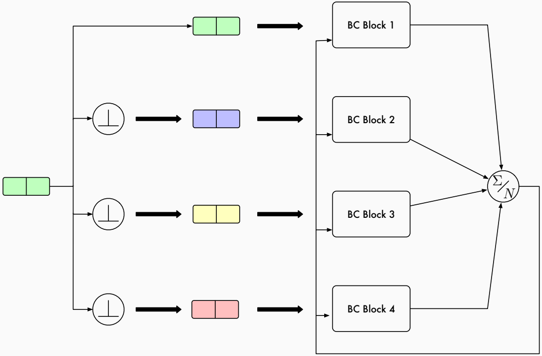
$$p \approx \frac{1}{T} \sum_{t=1}^{T} X_t$$

Parallelizing Stochastic Bitstreams



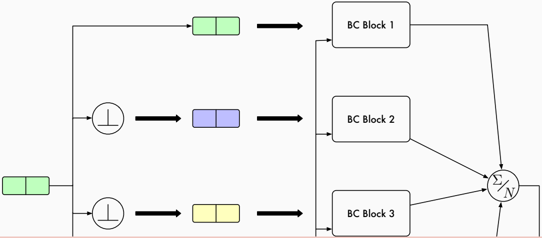
$$p \approx \frac{1}{T/N} \sum_{t=1}^{T/N} \frac{1}{N} \sum_{i=1}^{N} X_{t,i}$$

Parallelizing Stochastic Bitstreams



$$p \approx \frac{1}{T/N} \sum_{t=1}^{T/N} \frac{1}{N} \sum_{i=1}^{N} X_{t,i}$$

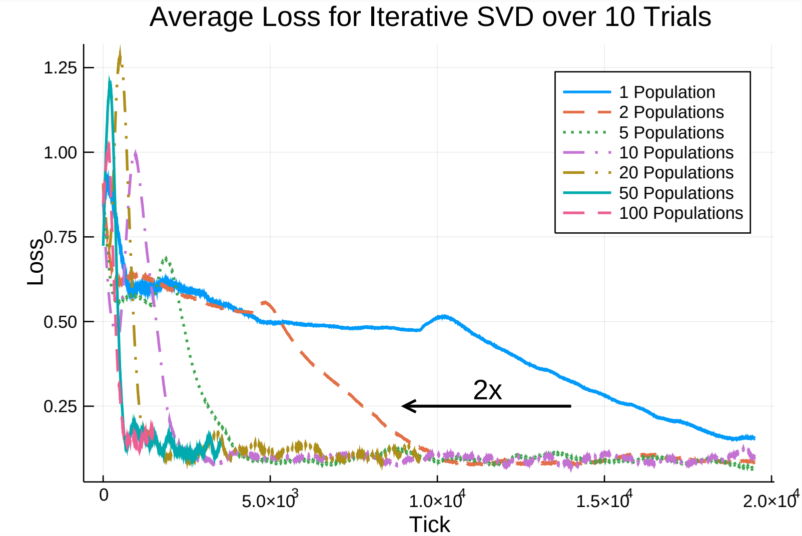
Parallelizing Stochastic Bitstreams



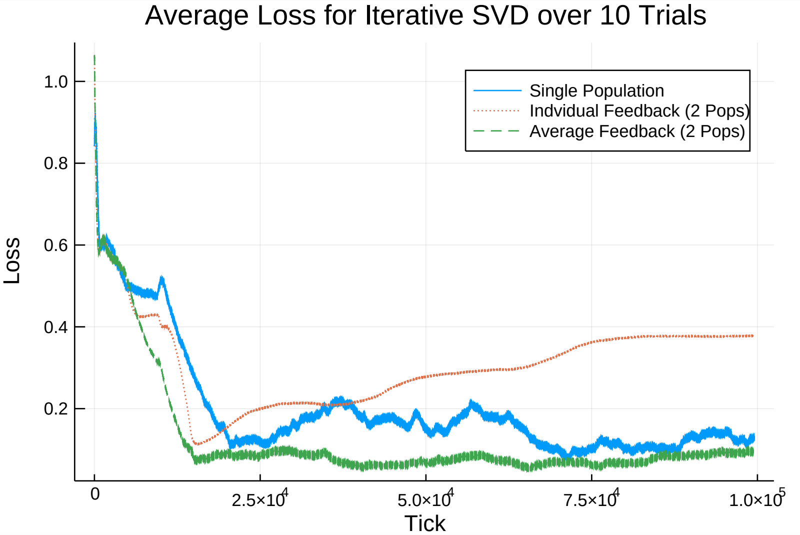
Theorem: For all interesting circuits, population coding guarantees that output can be fed back without decorrelation

$$p \approx \frac{1}{T/N} \sum_{t=1}^{T/N} \frac{1}{N} \sum_{i=1}^{N} X_{t,i}$$

Population Coding Experiments



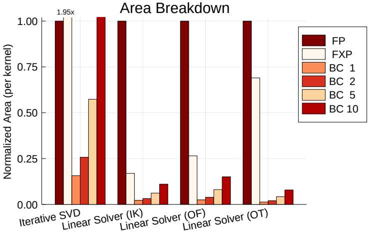
Population Coding Experiments

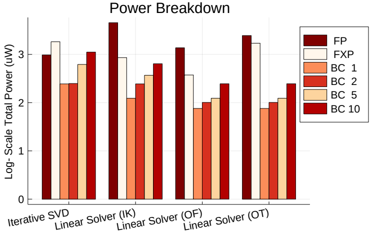


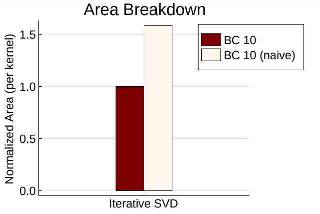
Population Coding in BitSAD

```
def loop(A: Matrix[SBitstream], v: Matrix[SBitstream]):
        (Matrix[SBitstream], Matrix[SBitstream], SBitstream)
        = populations(2, {
  // Update right singular vector
  var w = A * v
  var wScaled = w :/ math.sqrt(params.m)
  var u = wScaled / Matrix.norm(wScaled)

  // Update left singular vector
  var z = A.T * u
  var zScaled = z :/ math.sqrt(params.n)
  var sigma = Matrix.norm(zScaled)
  var _v = zScaled / sigma

  (u, _v, sigma)
})
```

Population Coding Results

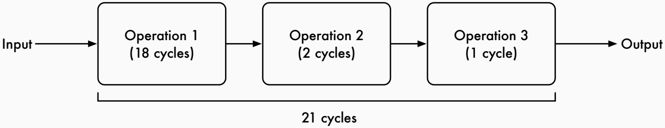




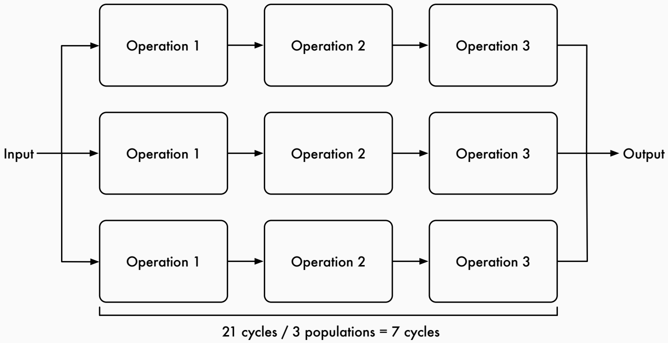


Optimizations for Stochastic Bitstreams

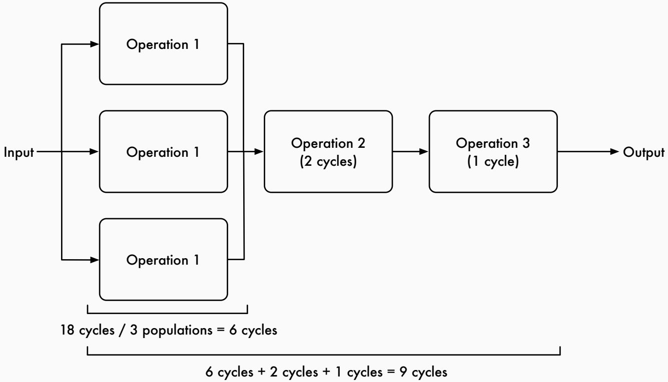
Population Coding Granularity



Population Coding Granularity



Population Coding Granularity



Preliminary Results of Granular Pop Coding

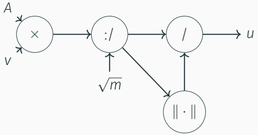
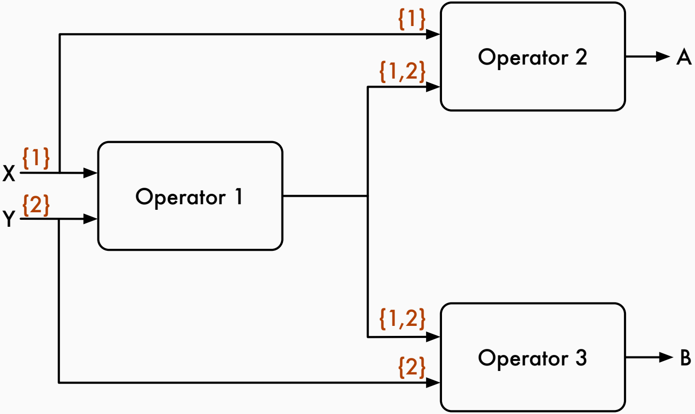


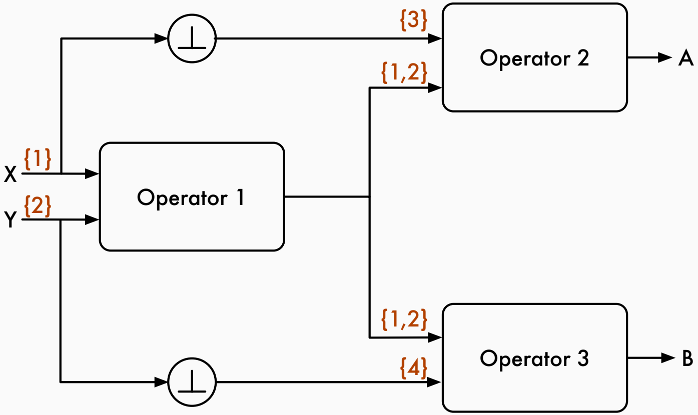
Table 1: A Comparison of Designs with Population Coding at Different Granularities

Variant	Average Cycles till Convergence	Area (# LUTs + # FFs)
8 Populations (Full Design)	19781	6807
8 Populations (Multiplier Only)	13613	4484

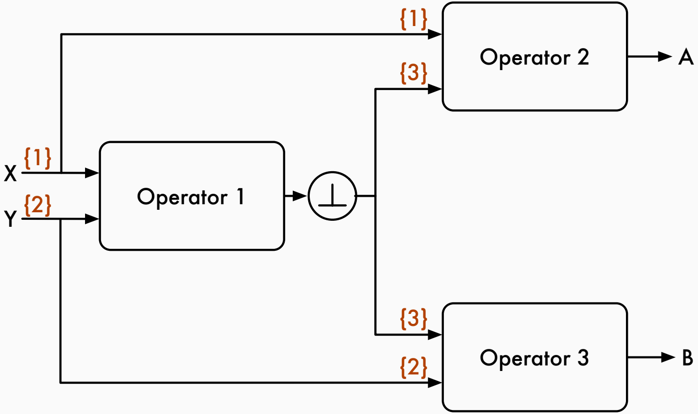
Optimal Placement of Decorrelators



Optimal Placement of Decorrelators



Optimal Placement of Decorrelators



Optimizations for Deterministic Bitstreams

Deterministic Bitstream Optimization

Consider the following digital state-variable filter:

```
// Get delay buffer values
val d1_old = delay1.pop
val d2_old = delay2.pop

// Update SDM outputs (f and q are compile time constants)
val d2 = sdm2.evaluate(f * d1_old + d2_old)
val d1 = sdm1.evaluate(f * (x - d2 - q * d1_old) + d1_old)

// Push new values into delay buffers
delay1.push(d1)
delay2.push(d2)
```

Deterministic Bitstream Optimization

Consider the following digital state-variable filter:

```
// Get delay buffer values
val d1_old = delay1.pop
val d2_old = delay2.pop

// Update SDM outputs (f and q are compile time constants)
val d2 = sdm2.evaluate(f * d1_old + d2_old)
val d1 = sdm1.evaluate(f * (x - d2 - q * d1_old) + d1_old)

// Push new values into delay buffers
delay1.push(d1)
delay2.push(d2)
```

With strength reduction:

```
val d2 = sdm2.evaluate(f * d1_old + d2_old)
val d1 = sdm1.evaluate(f * x - f * d2 - (f * q) * d1_old + d1_old)
```

Deterministic Bitstream Optimization

Consider the following digital state-variable filter:

```
// Get delay buffer values
val d1_old = delay1.pop
val d2_old = delay2.pop

// Update SDM outputs (f and q are compile time constants)
val d2 = sdm2.evaluate(f * d1_old + d2_old)
val d1 = sdm1.evaluate(f * (x - d2 - q * d1_old) + d1_old)

// Push new values into delay buffers
delay1.push(d1)
delay2.push(d2)
```

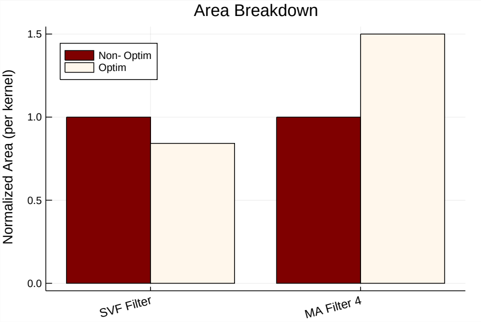
With strength reduction:

```
val d2 = sdm2.evaluate(f * d1_old + d2_old)
val d1 = sdm1.evaluate(f * x - f * d2 - (f * q) * d1_old + d1_old)
```

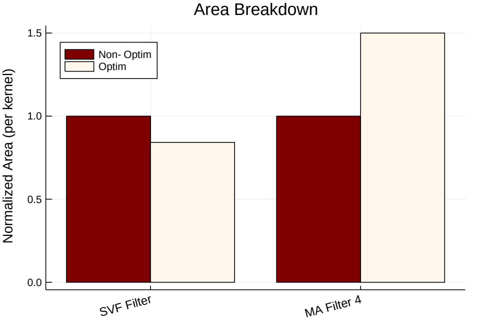
With algebraic simplification:

```
val d2 = sdm2.evaluate(f * d1_old + d2_old)
val d1 = sdm1.evaluate(f * x - f * d2 + (1 - f * q) * d1_old)
```

Deterministic Bitstream Optimization



Deterministic Bitstream Optimization



Synthesize submodules with standard cell library and provide area estimates to compiler as a clue

Concluding Remarks

Conclusion

Our language, BitSAD, allows users to:

1. design algorithms at a high level
2. simulate bit-level, cycle-accurate results
3. generate hardware automatically

We also introduced:

1. population coding to parallelize stochastic computing circuits w/o sacrificing accuracy
2. optimizations for deciding population coding granularity and decorrelator placement
3. optimizations for deterministic bitstream designs

Questions?

Check out BitSAD on GitHub:

`https://github.com/UW-PHARM/BitSAD`
`https://github.com/UW-PHARM/BitBench`

References i

- Dllu (2017). URL: https://commons.wikimedia.org/wiki/File:Waymo_Chrysler_Pacifica_in_Los_Altos,_2017.jpg.
- Dubrofsky, Elan (2009). "Homography Estimation". PhD thesis. Carleton University.
- Ma, Kevin Y. (2015). *RoboBee*. URL: http://www.aboutkevinma.com/index.html#publications (visited on 04/01/2018).

Malis, Ezio and Manuel Vargas (2007). "Deeper understanding of the homography decomposition for vision-based control". In: *Sophia* 6303.6303, p. 90. ISSN: 0036-8075. DOI: 10.1126/science.318.5857.1691b. URL: http://hal.archives-ouvertes.fr/inria-00174036/.