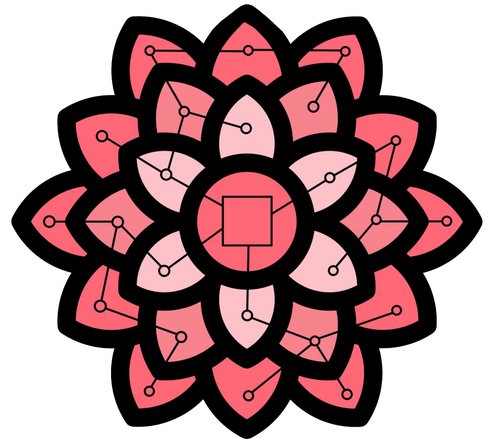
Predictable Accelerator Design

with Time-Sensitive Affine Types



Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer,
Yuwei Ye, Apurva Koti, Adrian Sampson, Zhiru Zhang

capra.cs.cornell.edu/dahlia

Computer **A**rchitecture and
Programming **A**bstractions Group

Rachit Nigam

- Second year PhD
- Computer Architect by day
- Programming languages by night

Rachit Nigam

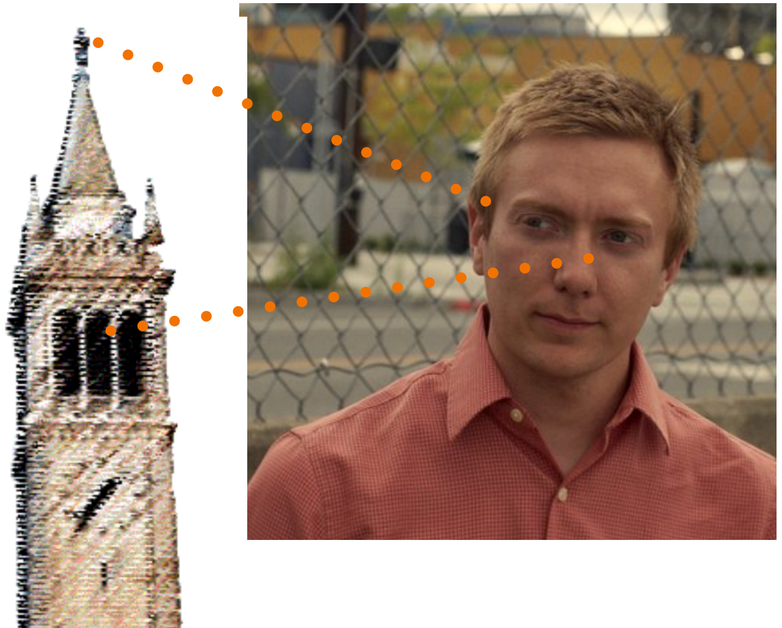
- Second year PhD
- Computer Architect by day
- Programming languages by night

Adrian Sampson

- Nth year PhD
- PL & Compilers & Architecture

Rachit Nigam

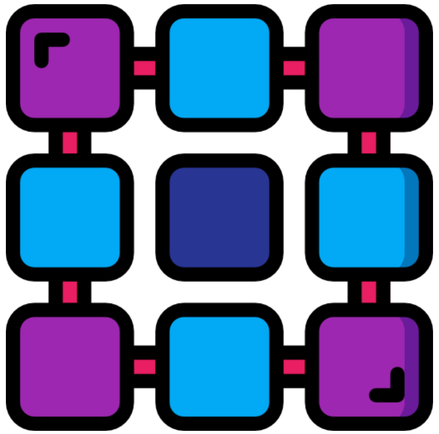
- Second year PhD
- Computer Architect by day
- Programming languages by night

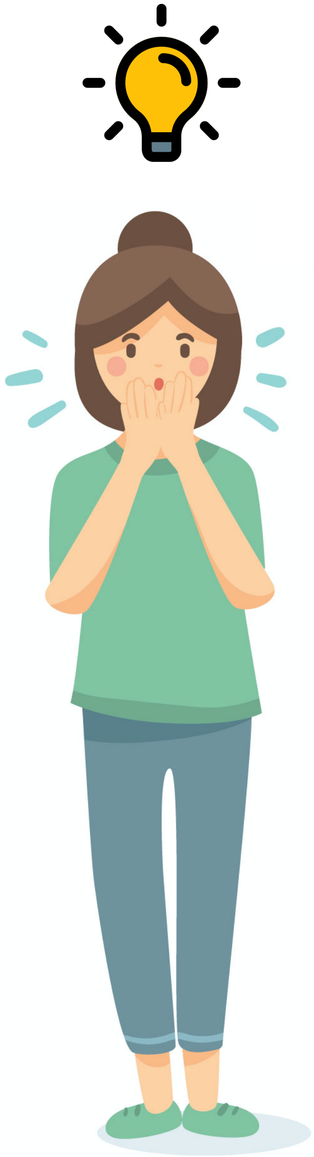


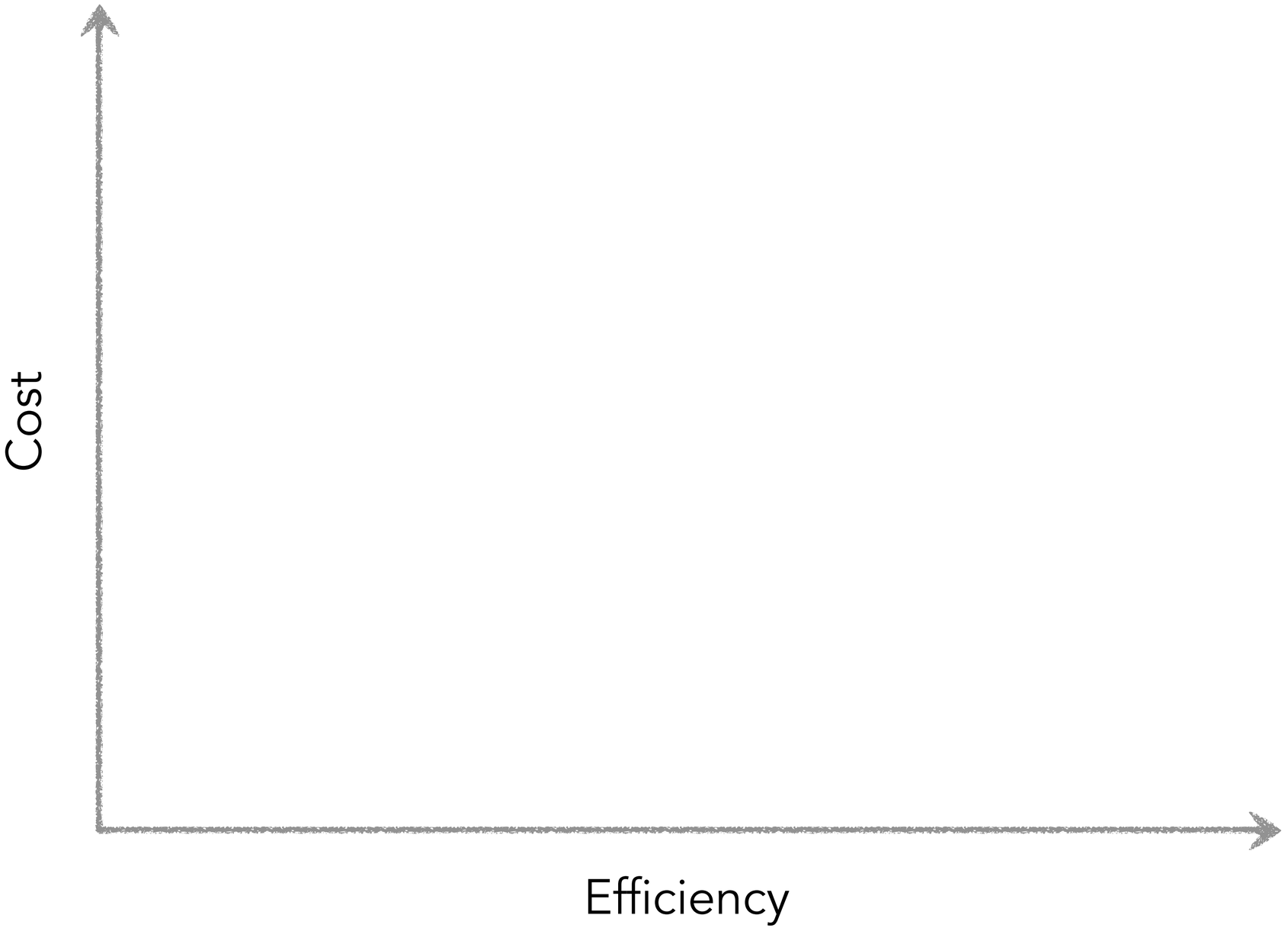
Adrian Sampson

- Nth year PhD
- PL & Compilers & Architecture

Icons made by Smashicons






Cost
Efficiency

Cost

CPU

Efficiency

Cost

GPU

CPU

Efficiency

Cost

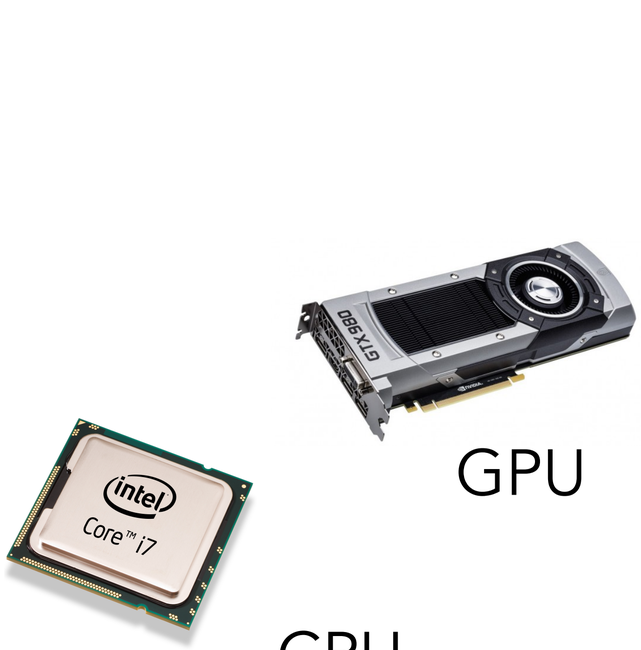
ASIC

GPU

CPU

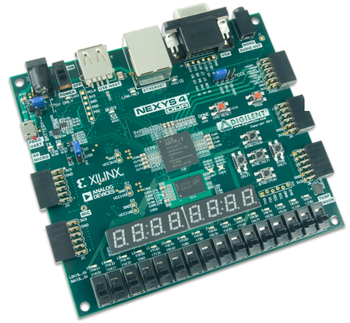
Efficiency

Cost



CPU

GPU



FPGA

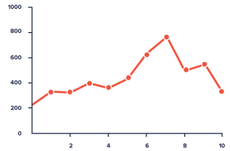
ASIC

Efficiency

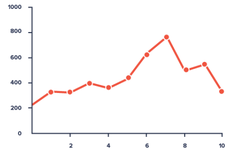
Ada's Journey

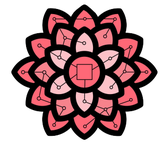
Ada's Journey

Ada's Journey

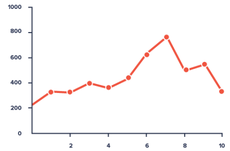


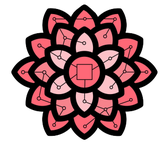
Ada's Journey





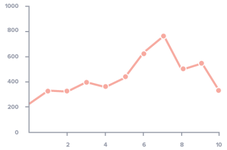
Ada's Journey

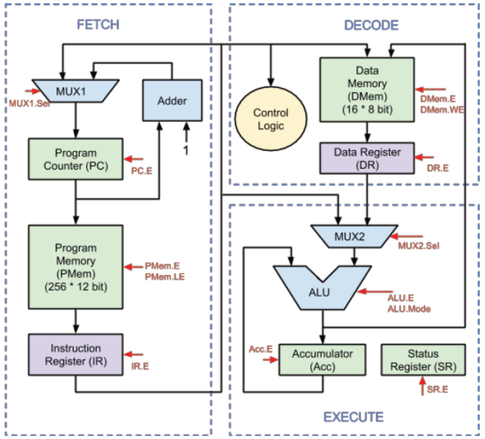




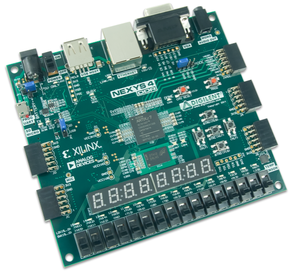
Our Research

Ada's Journey

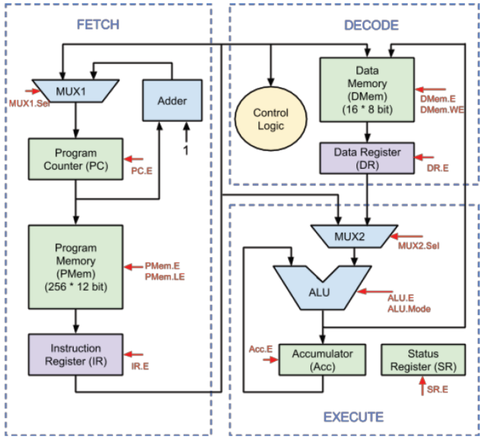




Hardware Design



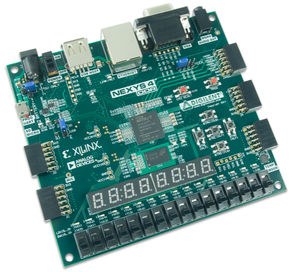
FPGA



Hardware Design

SystemVerilog

Traditional HDLs



FPGA

Traditional HDLs

```
module ctr (input            up_down,
                clk,
                rstn,
            output reg [2:0]   out);

  always @ (posedge clk)
    if (!rstn)
      out <= 0;
    else begin
      if (up_down)
        out <= out + 1;
      else
        out <= out - 1;
    end
endmodule
```

Traditional HDLs

```
module ctr (input            up_down,
                clk,
                rstn,
            output reg [2:0]   out);

  always @ (posedge clk)
    if (!rstn)
      out <= 0;
    else begin
      if (up_down)
        out <= out + 1;
      else
        out <= out - 1;
    end
endmodule
```

- **Time** is real

Traditional HDLs

```
module ctr (input            up_down,
                clk,
                rstn,
            output reg [2:0]   out);

  always @ (posedge clk)
    if (!rstn)
      out <= 0;
    else begin
      if (up_down)
        out <= out + 1;
      else
        out <= out - 1;
    end
endmodule
```

- Time is real
- **Concurrent** semantics

Traditional HDLs

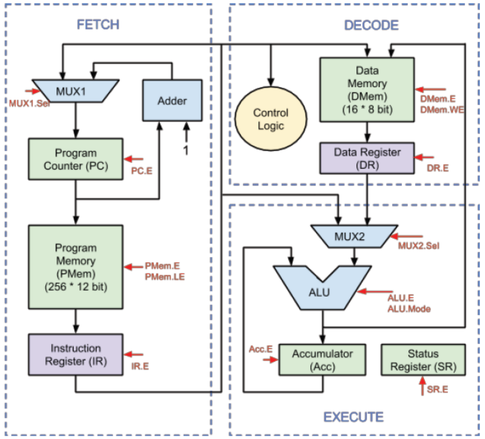
```
neptune_metadata_unpack #(
    .TUSER_W (TUSER_W)
) neptune_metadata_unpack_inst (
    // input
    .metadata        (s_axis_tuser),
    // output
    .std_md          (neptune_std_md),
    .mem_instr_0     (),
    .mem_instr_1     (),
    .mem_instr_2     (),
    .egress_ts       (),
    .fpm             (fpm),
    .reserved        ()
);
```

- Time is real
- Concurrent semantics

Traditional HDLs

```
neptune_metadata_unpack #(
    .TUSER_W (TUSER_W)
) neptune_metadata_unpack_inst (
    // input
    .metadata          (s_axis_tuser),
    // output
    .std_md            (neptune_std_md),
    .mem_instr_0       (),
    .mem_instr_1       (),
    .mem_instr_2       (),
    .egress_ts         (),
    .fpm               (fpm),
    .reserved          ()
);
```

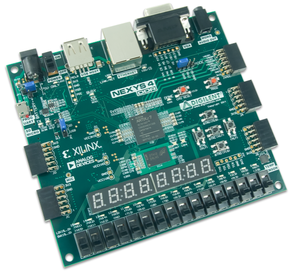
- Time is real
- Concurrent semantics
- **Wires** and hardware modules



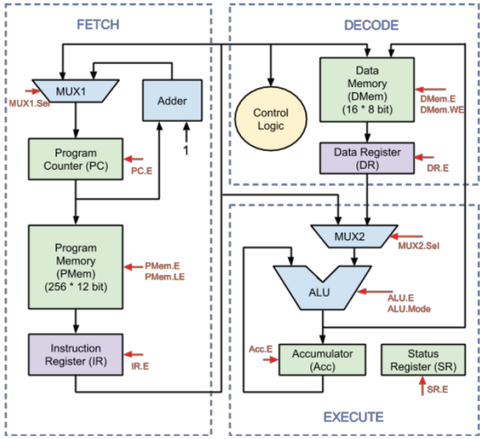
Hardware Design

SystemVerilog

Traditional HDLs



FPGA



Hardware Design

SystemVerilog

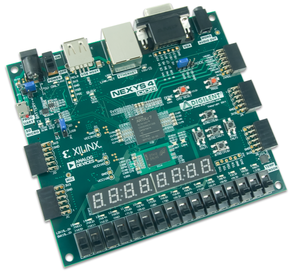
Traditional HDLs

bluespec

CHISEL

PyMTL

Modern HDLs



FPGA

bluespec

CHISEL

PyMTL

Modern HDLs

```
// Generalized FIR filter parameterized by the convolution coefficients
class FirFilter(bitWidth: Int, coeffs: Seq[UInt]) extends Module {
  val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W)))
  zs(0) := io.in
  for (i <- 1 until coeffs.length) {
    zs(i) := zs(i-1)
  }

  // Do the multiplies
  val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i))

}
```

bluespec

CHISEL

PyMTL

Modern HDLs

```
// Generalized FIR filter parameterized by the convolution coefficients
class FirFilter(bitWidth: Int, coeffs: Seq[UInt]) extends Module {









  val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W)))
  zs(0) := io.in
  for (i <- 1 until coeffs.length) {
    zs(i) := zs(i-1)
  }

  // Do the multiplies
  val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i))



}
```

High level constructs!

bluespec

CHISEL

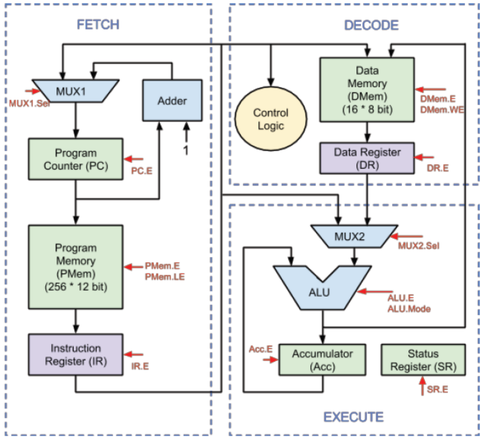
PyMTL

Modern HDLs

```
// Generalized FIR filter parameterized by the convolution coefficients
class FirFilter(bitWidth: Int, coeffs: Seq[UInt]) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt(bitWidth.W))
  })
  // Create the serial-in, parallel-out shift register
  val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W)))
  zs(0) := io.in
  for (i <- 1 until coeffs.length) {
    zs(i) := zs(i-1)
  }

  // Do the multiplies
  val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i))

  // Sum up the products
  io.out := products.reduce(_ + _)
}
```

High level constructs!

… **elaborated** into circuits



Hardware Design

SystemVerilog

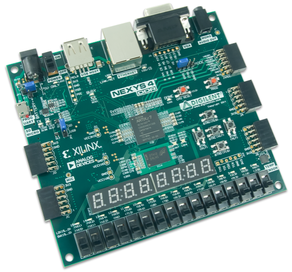
Traditional HDLs

bluespec

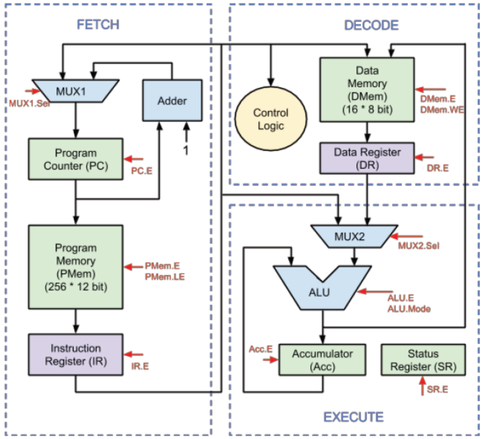
CHISEL

PyMTL

Modern HDLs



FPGA



Hardware Design

High-Level Synthesis

Traditional HDLs

Modern HDLs

FPGA

High-Level
Synthesis

```
int m1[512], m2[512], sum;
for (int i = 0; i < 512; i++) {
  sum += m1[i] * m2[i]
}
```

High-Level Synthesis

```
int m1[512], m2[512], sum;
for (int i = 0; i < 512; i++) {
  sum += m1[i] * m2[i]
}
```

Software programs that are

compiled to hardware designs

High-Level Synthesis

Level of abstraction

bluespec

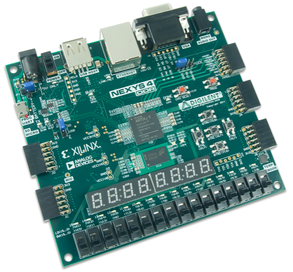
CHISEL

PyMTL

Modern HDLs

SystemVerilog

Traditional HDLs



FPGA

High-Level Synthesis

Level of abstraction

bluespec

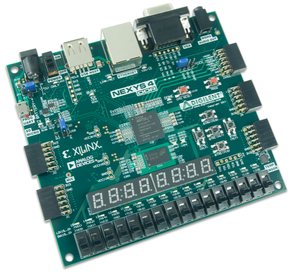
CHISEL

PyMTL

Modern HDLs

SystemVerilog

Traditional HDLs



FPGA

Circuit

Specification

Computational
Specification

High-Level Synthesis

Level of abstraction

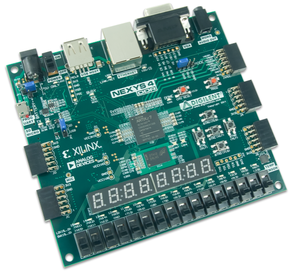
bluespec
CHISEL
PyMTL

Modern HDLs

Circuit
Specification

SystemVerilog

Traditional HDLs



FPGA

loops

method calls

arrays

break

malloc

functions

conditionals

loops

~~method calls~~

arrays

~~break~~

~~malloc~~

functions*

conditionals

loops

~~method calls~~

arrays

~~break~~

~~malloc~~

functions*

conditionals

loops

~~method calls~~

arrays

~~break~~

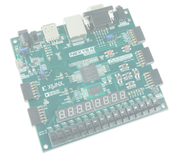
~~malloc~~

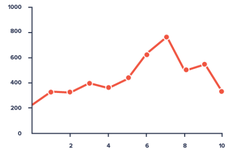
functions*

conditionals

High-performance
hardware designs

Ada's Journey





Super secret™ accelerator

Hardware

Super secret™ accelerator

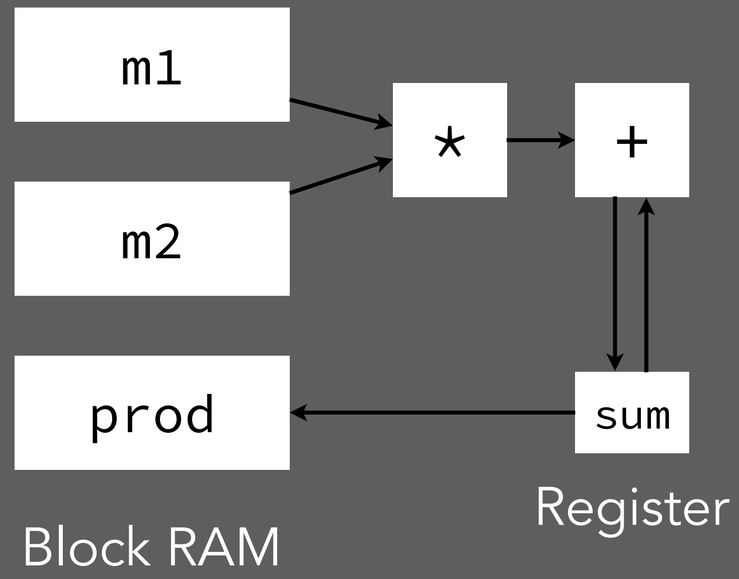
```
int m1[512][512];
int m2[512][512];
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

Hardware

Super secret™ accelerator

```
int m1[512][512];
int m2[512][512];
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

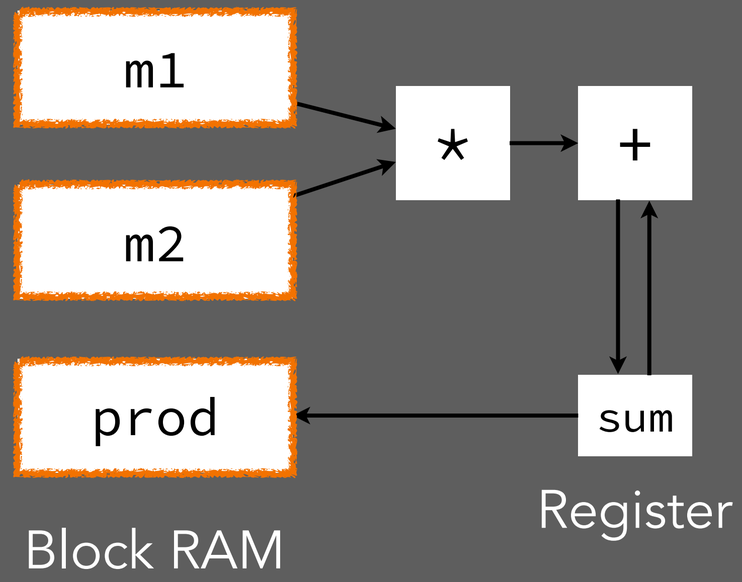
Hardware



Super secret™ accelerator

```
int m1[512][512];
int m2[512][512];
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

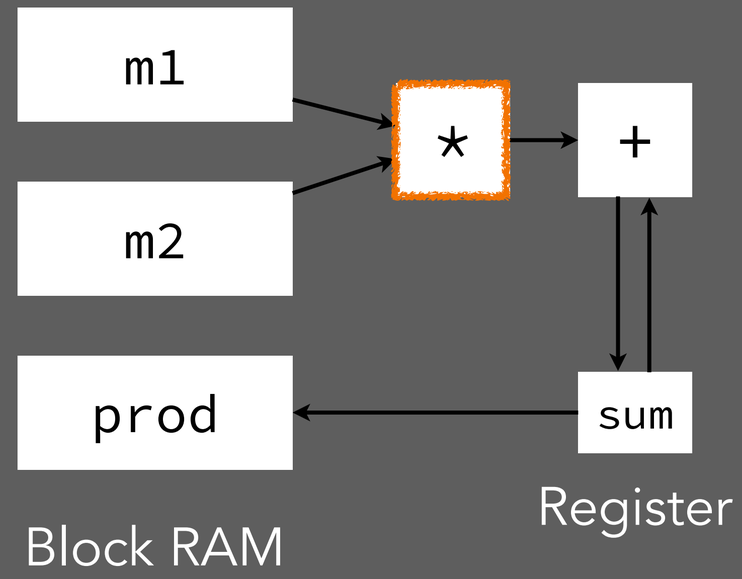
Hardware



Super secret™ accelerator

```
int m1[512][512];
int m2[512][512];
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

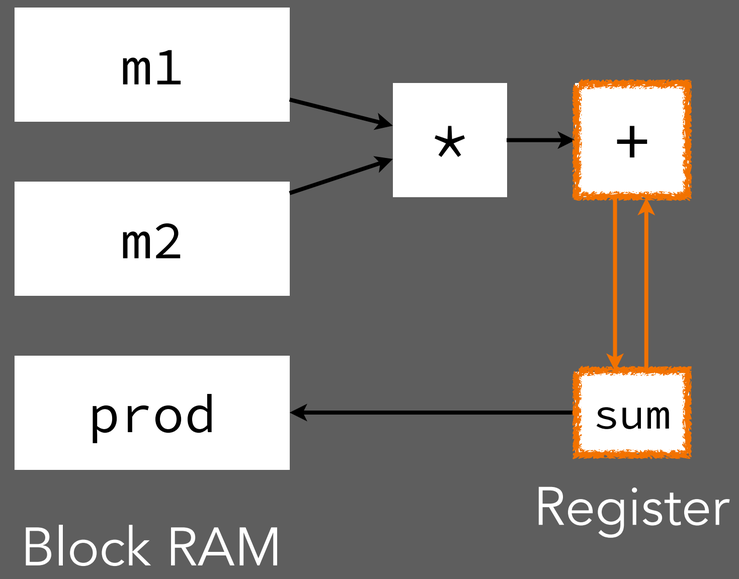
Hardware



Super secret™ accelerator

```
int m1[512][512];
int m2[512][512];
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

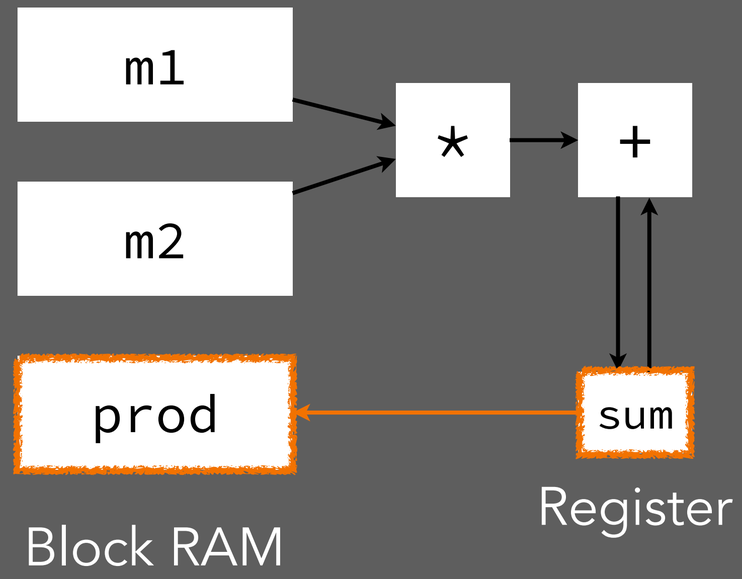
Hardware



Super secret™ accelerator

```
int m1[512][512];
int m2[512][512];
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

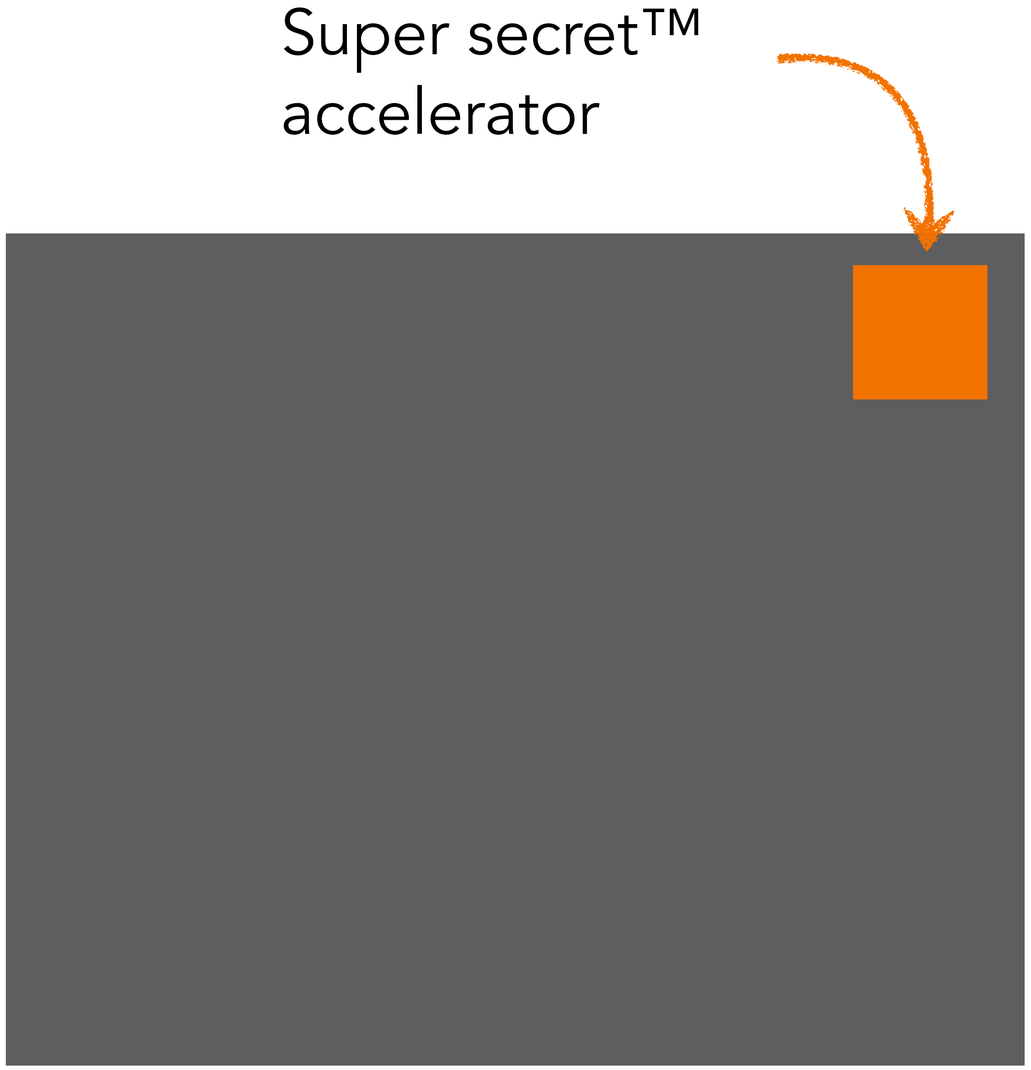
Hardware

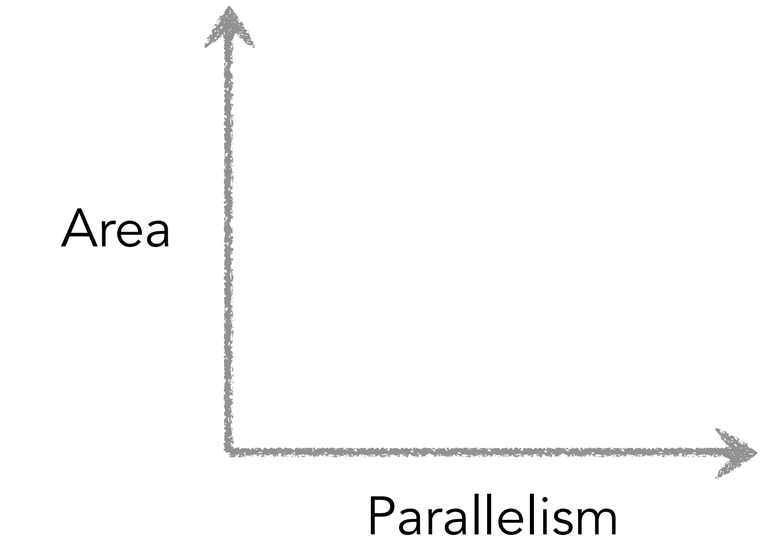


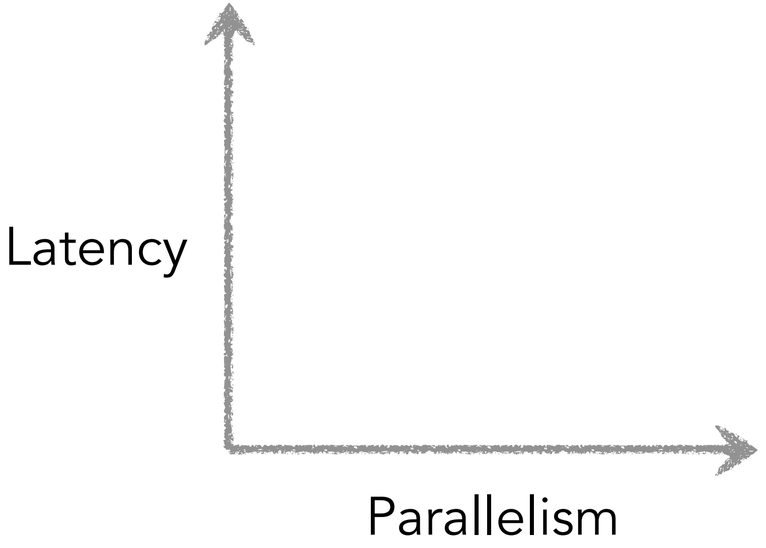
reaching up to **35.9 effective TFLOPS** for a large GRU over hundreds of timesteps. This represents an approximate two orders of magnitude advantage over the Titan Xp. This is

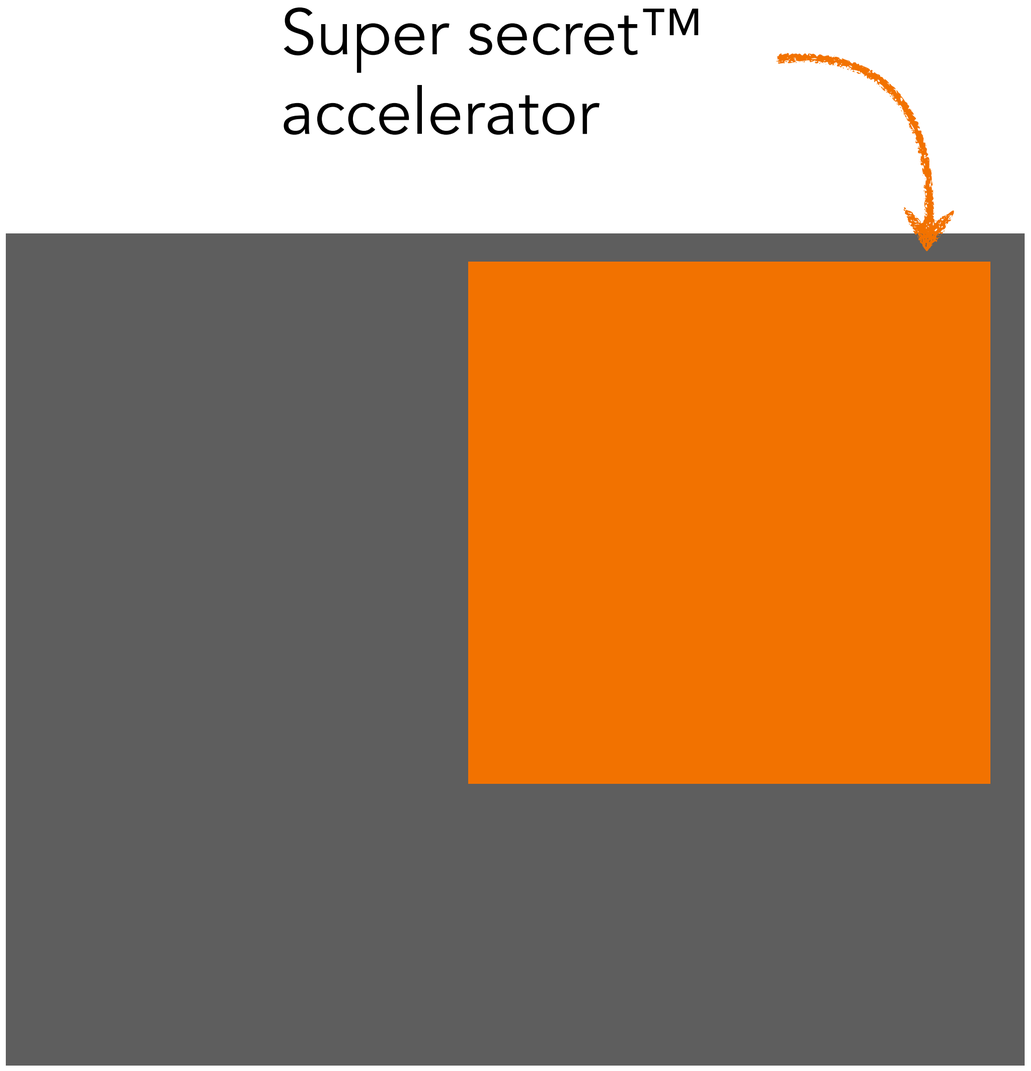
Darwin: A Genomics Co-processor Provides up to 15,000× acceleration on long read assembly

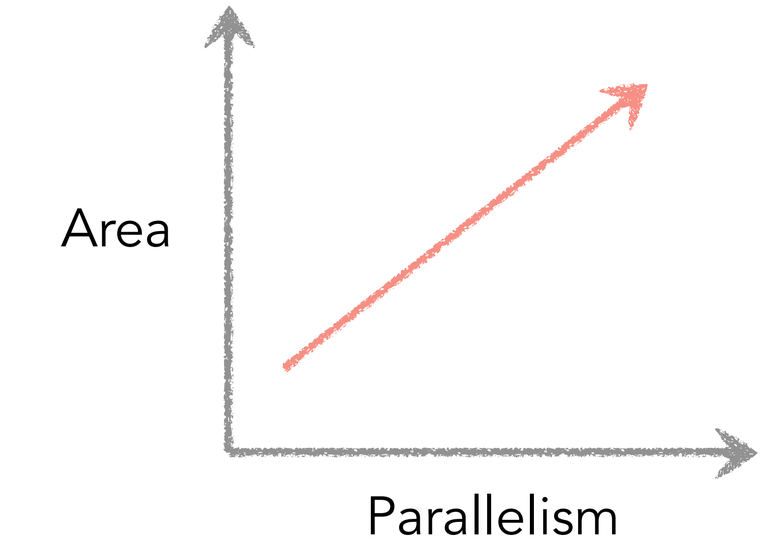
ranking candidate documents. Under high load, the large-scale reconfigurable fabric improves the ranking throughput of each server by a factor of 95% for a fixed latency distribution—or, while maintaining equivalent throughput, reduces the tail latency by 29%.

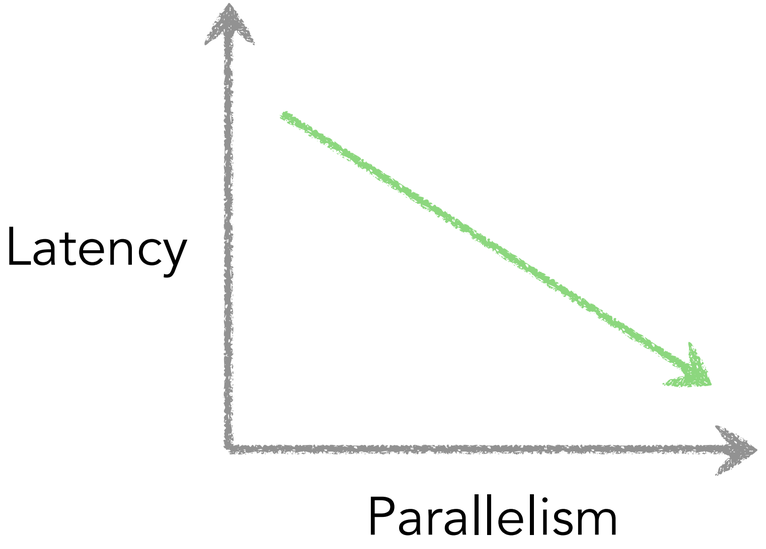

Super secret™
accelerator
Resources on the FPGA


Area
Parallelism


Latency
Parallelism


Super secret™
accelerator
Resources on the FPGA


Area
Parallelism


Latency
Parallelism

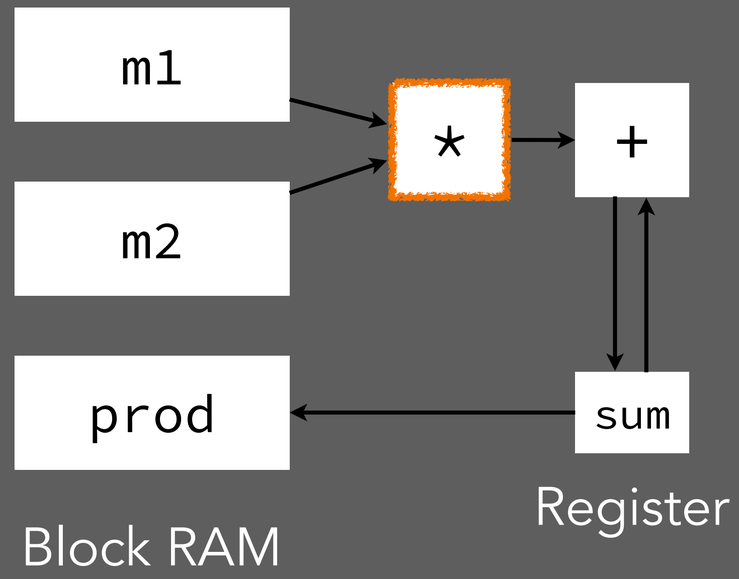
Area-Latency trade-offs

Create more processing elements to extract parallelism

Super secret™ accelerator

```
int m1[512][512];
int m2[512][512];
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

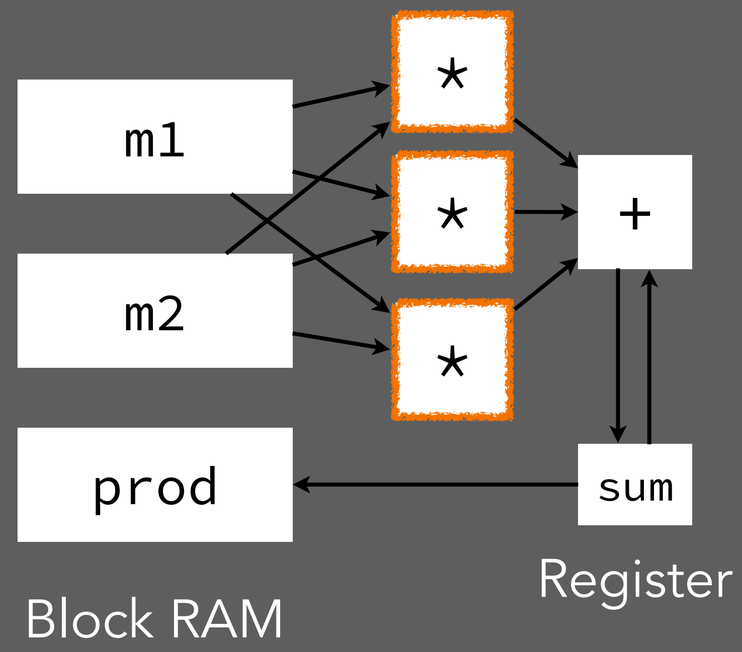
Hardware



Super secret™ accelerator

```
int m1[512][512];
int m2[512][512];
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

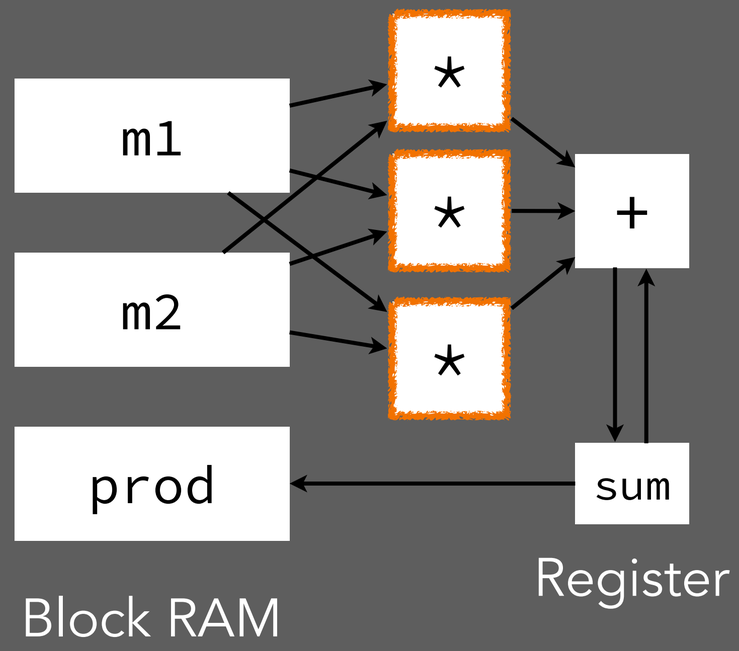
Hardware

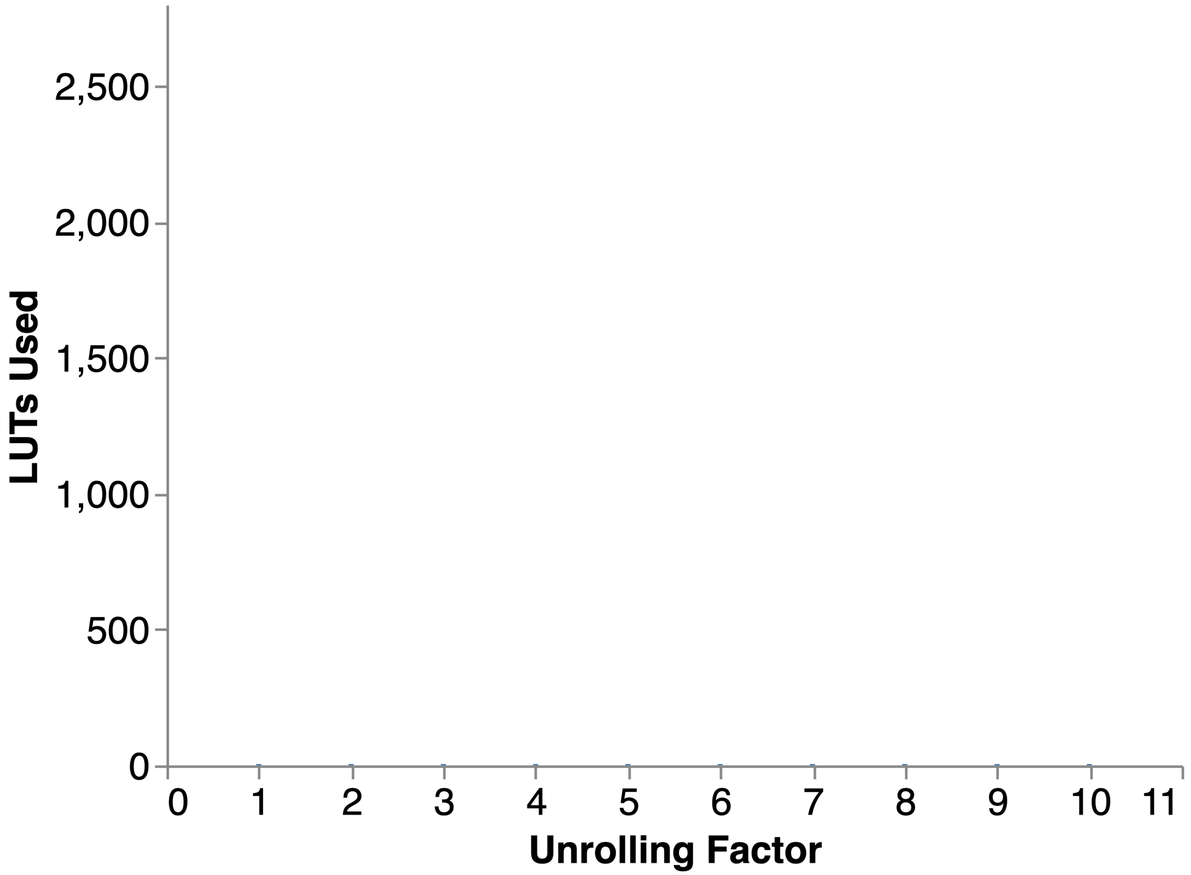


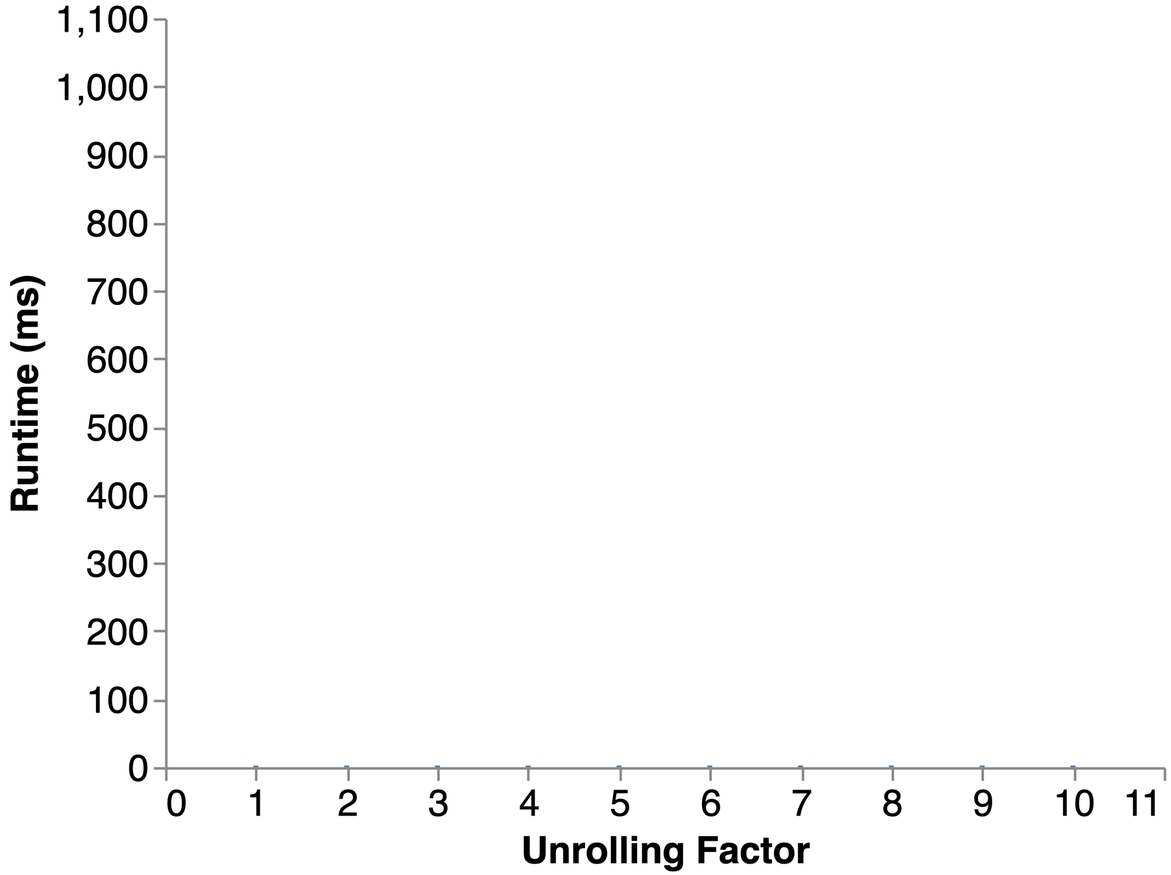
Super secret™ accelerator

```
int m1[512][512];
int m2[512][512];
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      #pragma HLS UNROLL factor=3
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

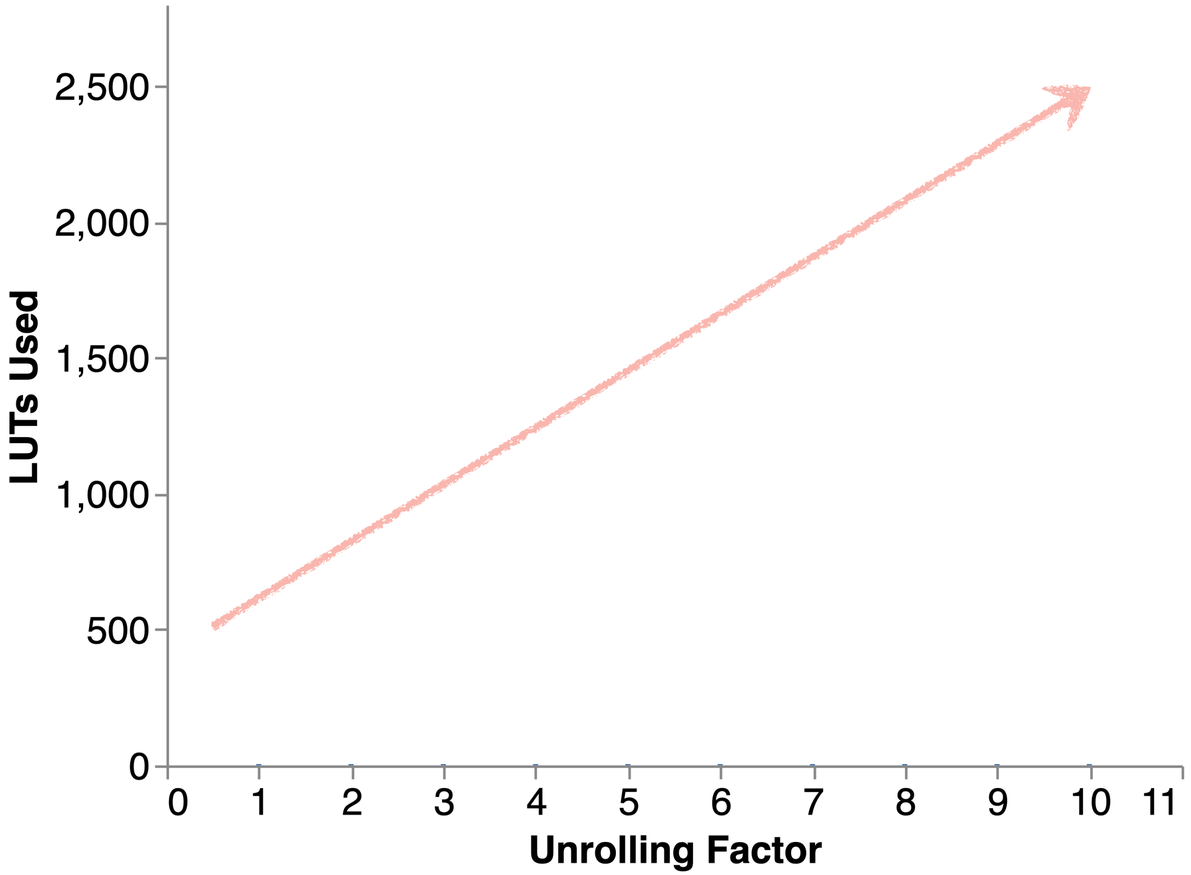
Hardware

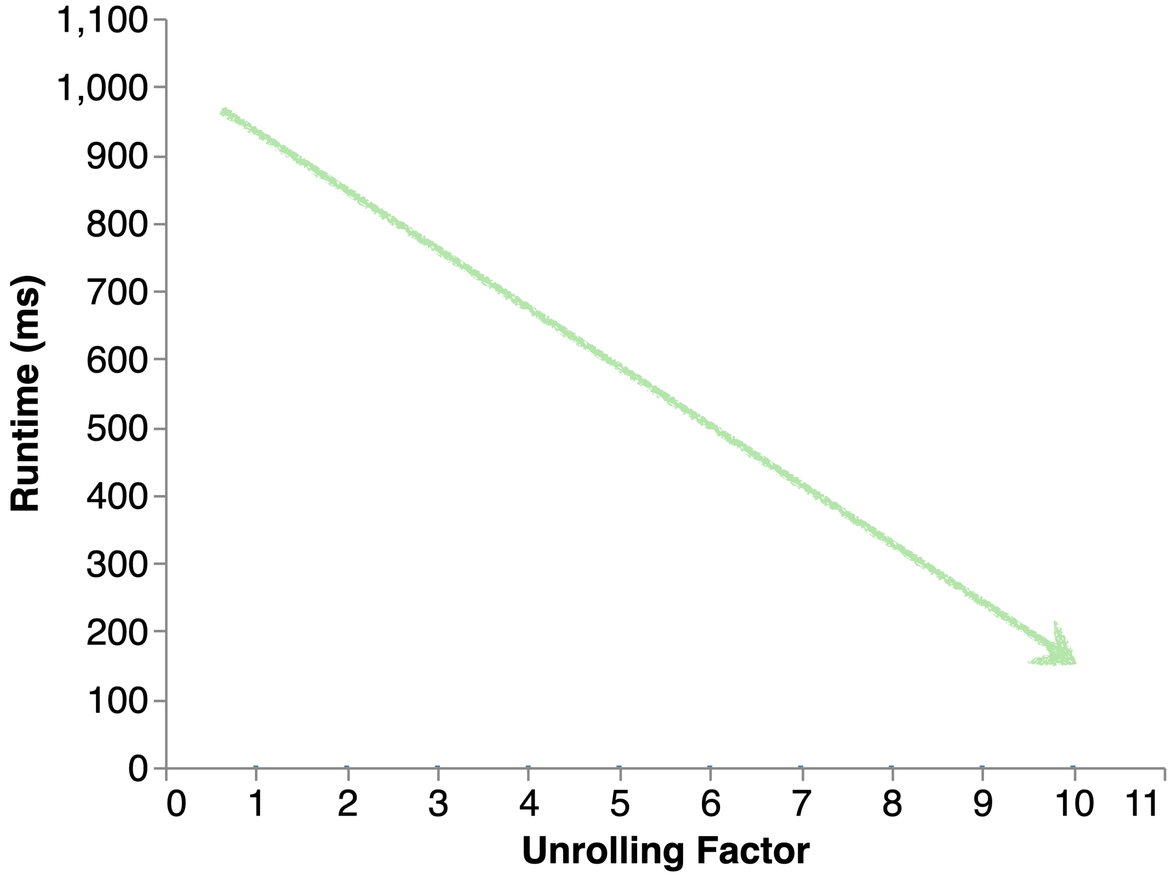




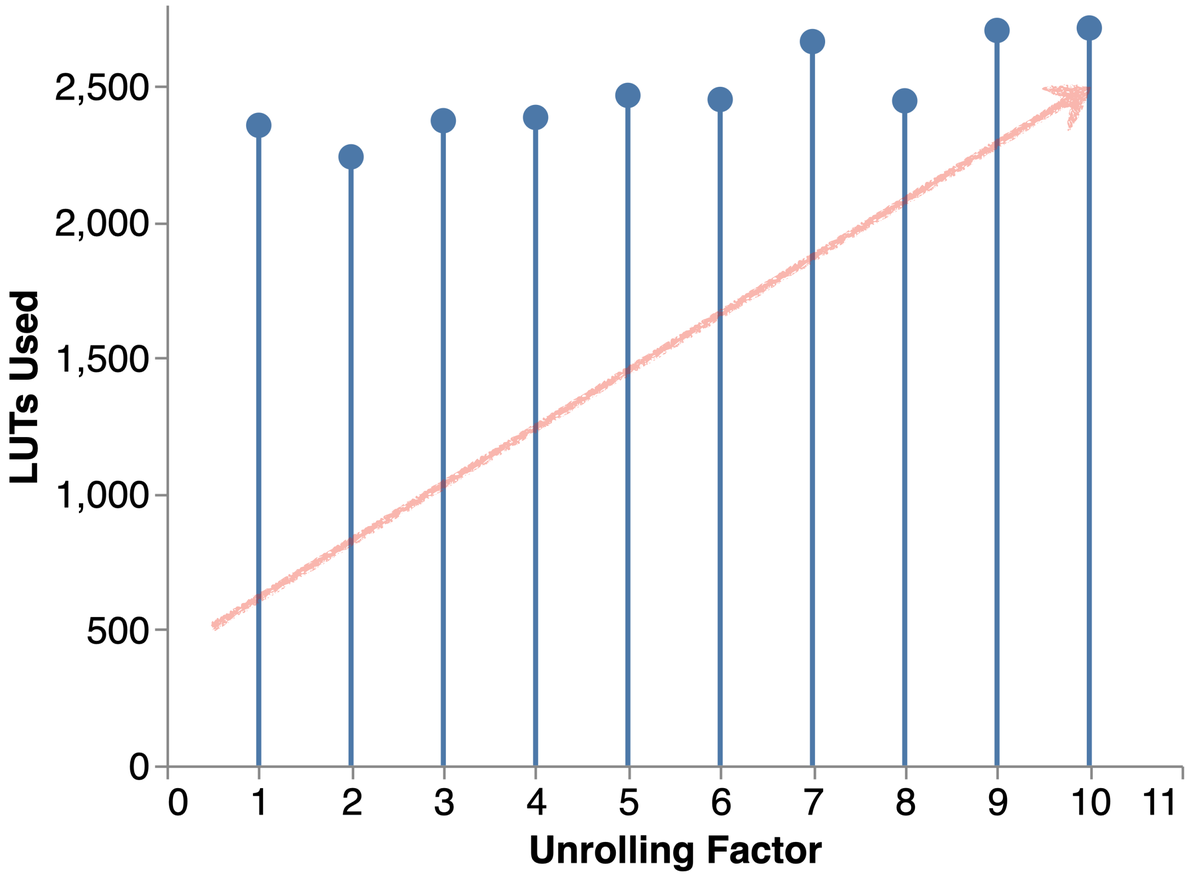


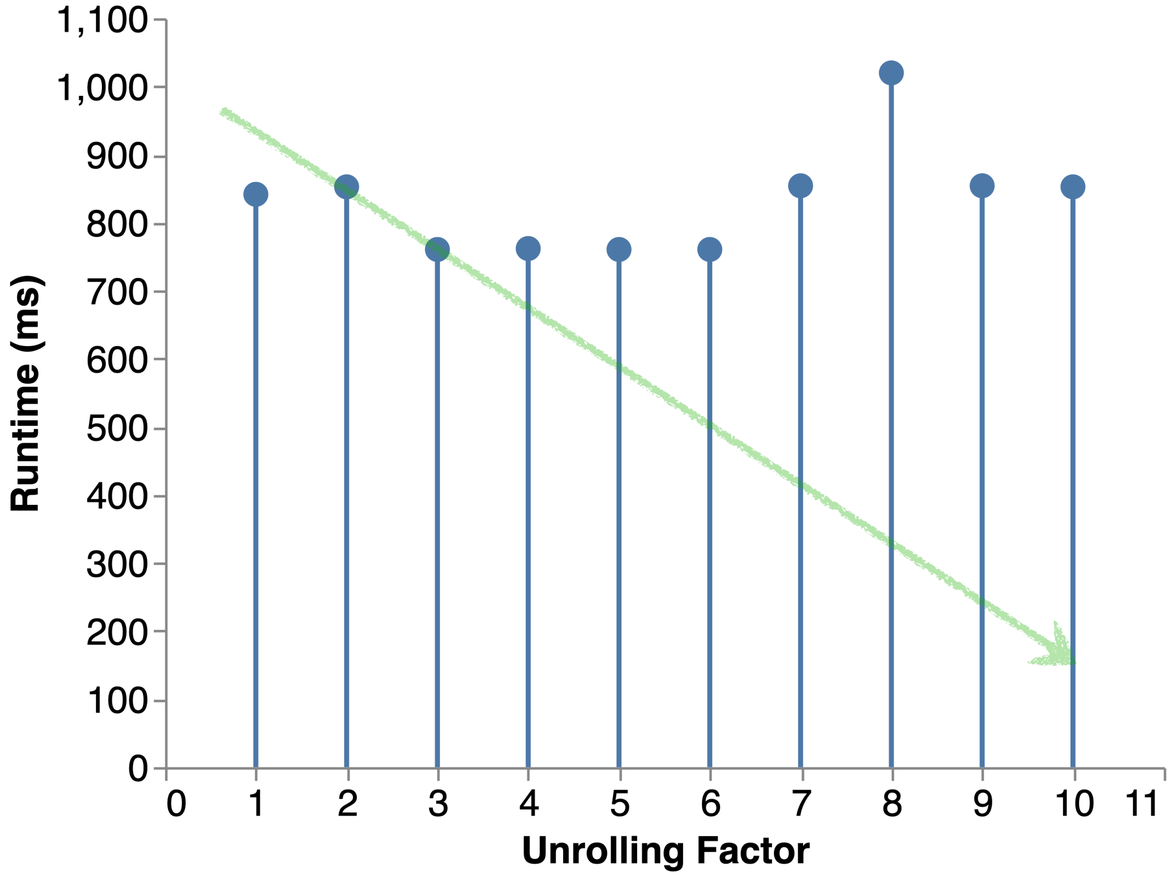
Area-Latency trade-offs





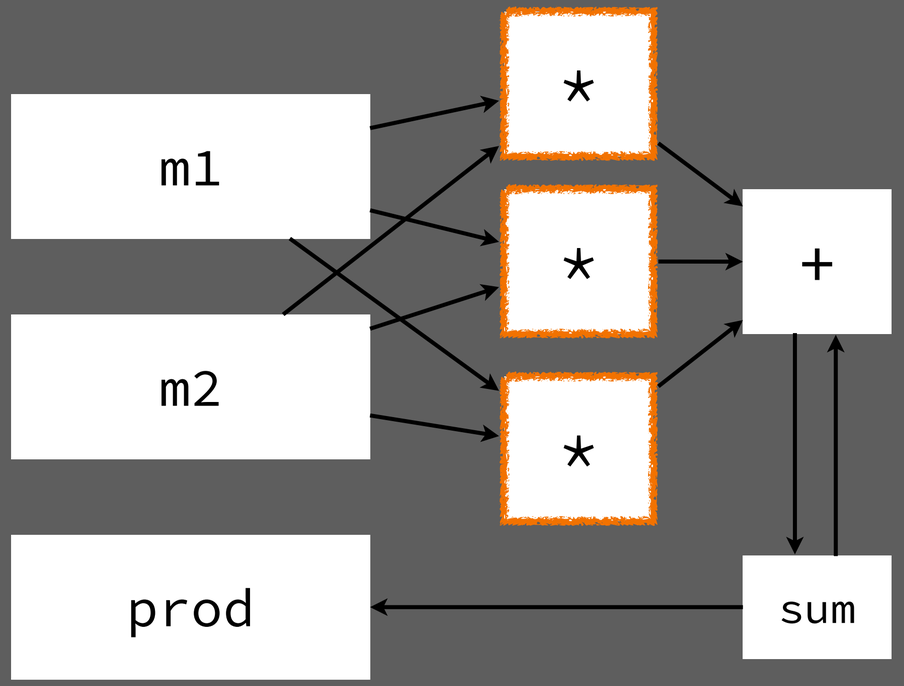
Area-Latency trade-offs



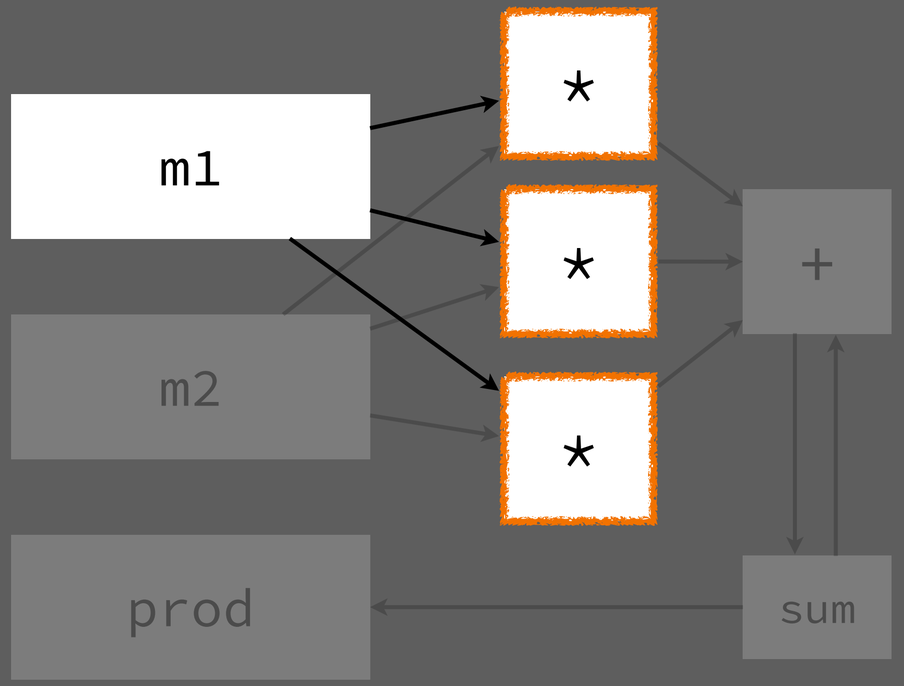


Area-Latency trade-offs

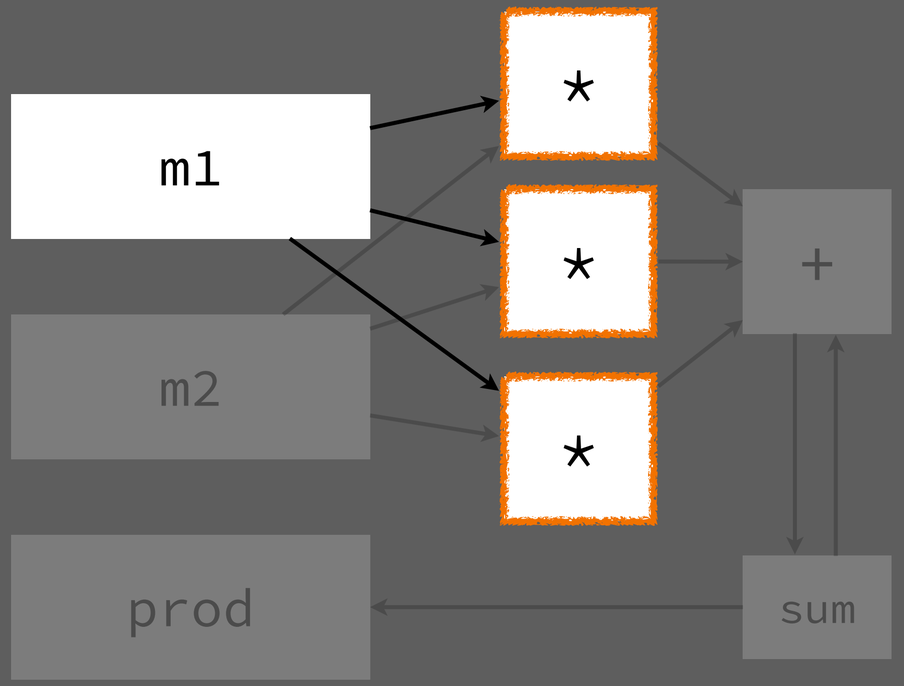
Hardware



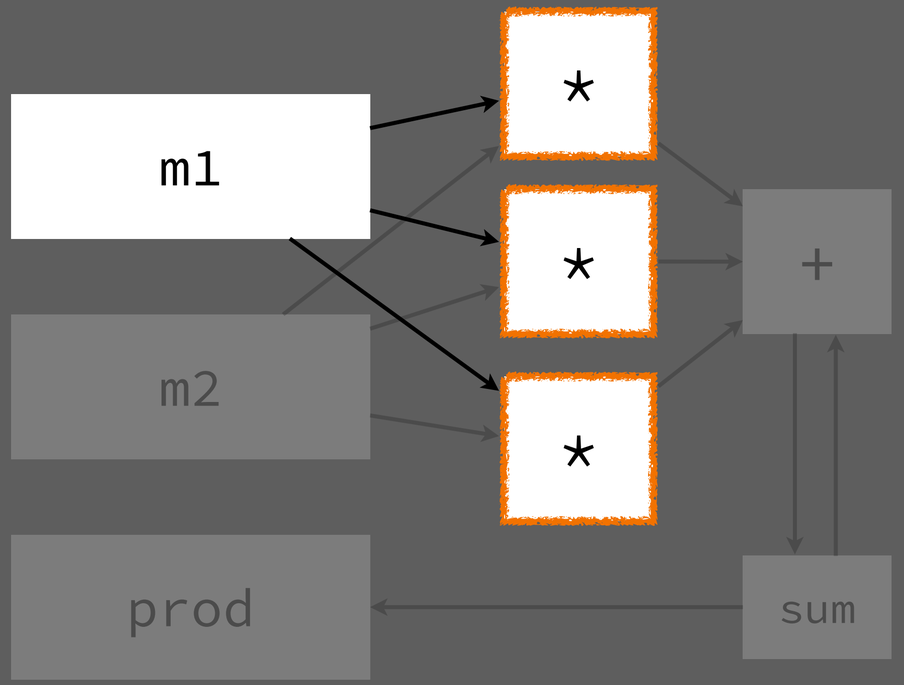
Hardware



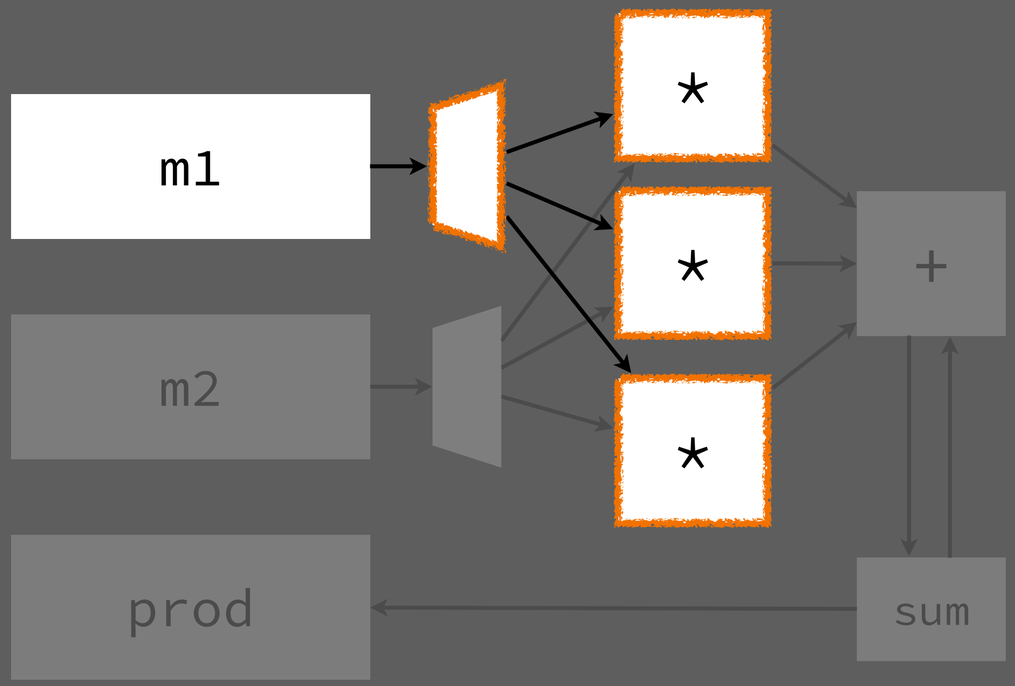
Hardware



Hardware

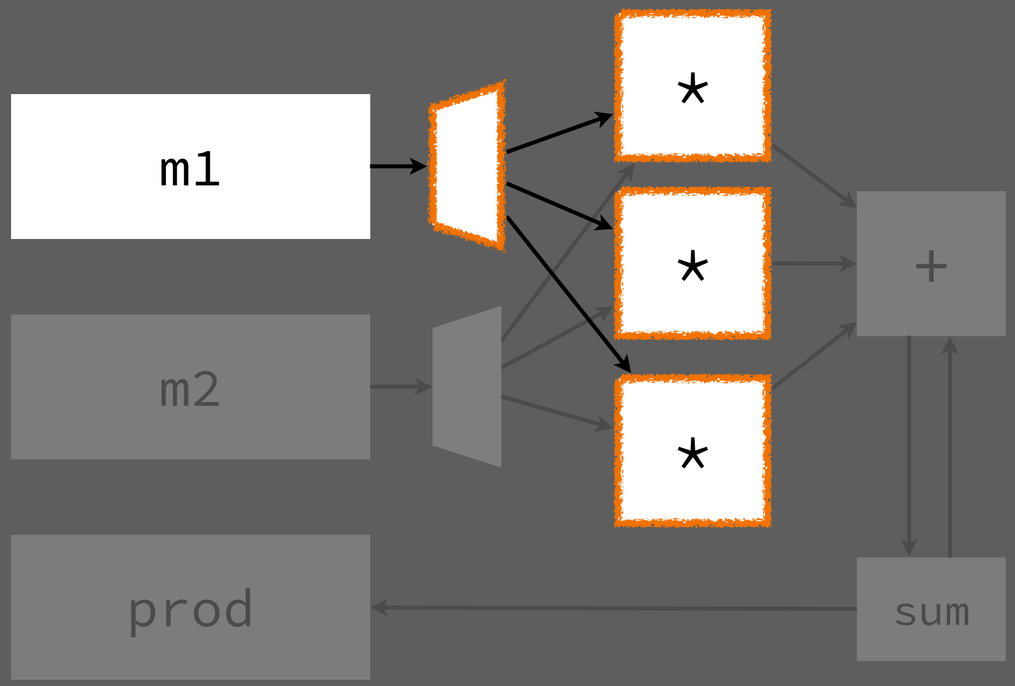


Hardware

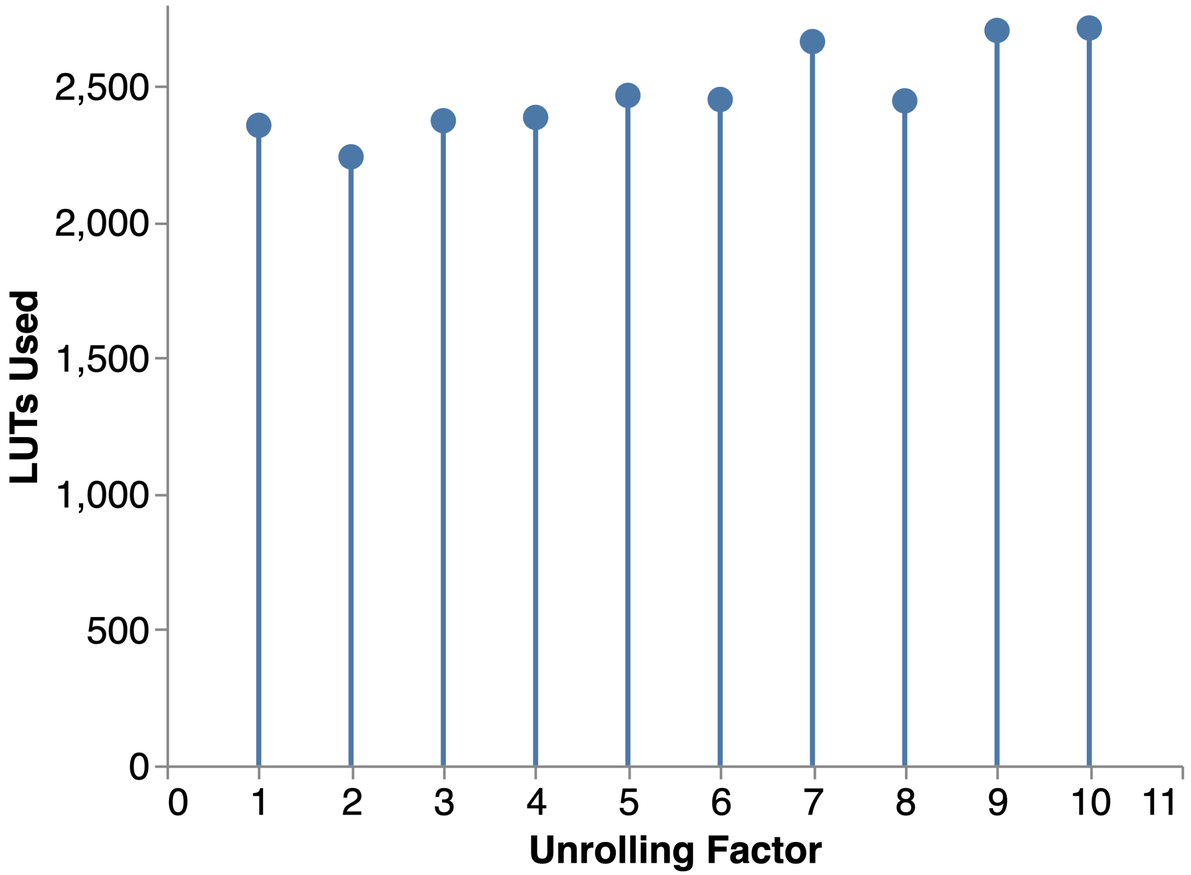


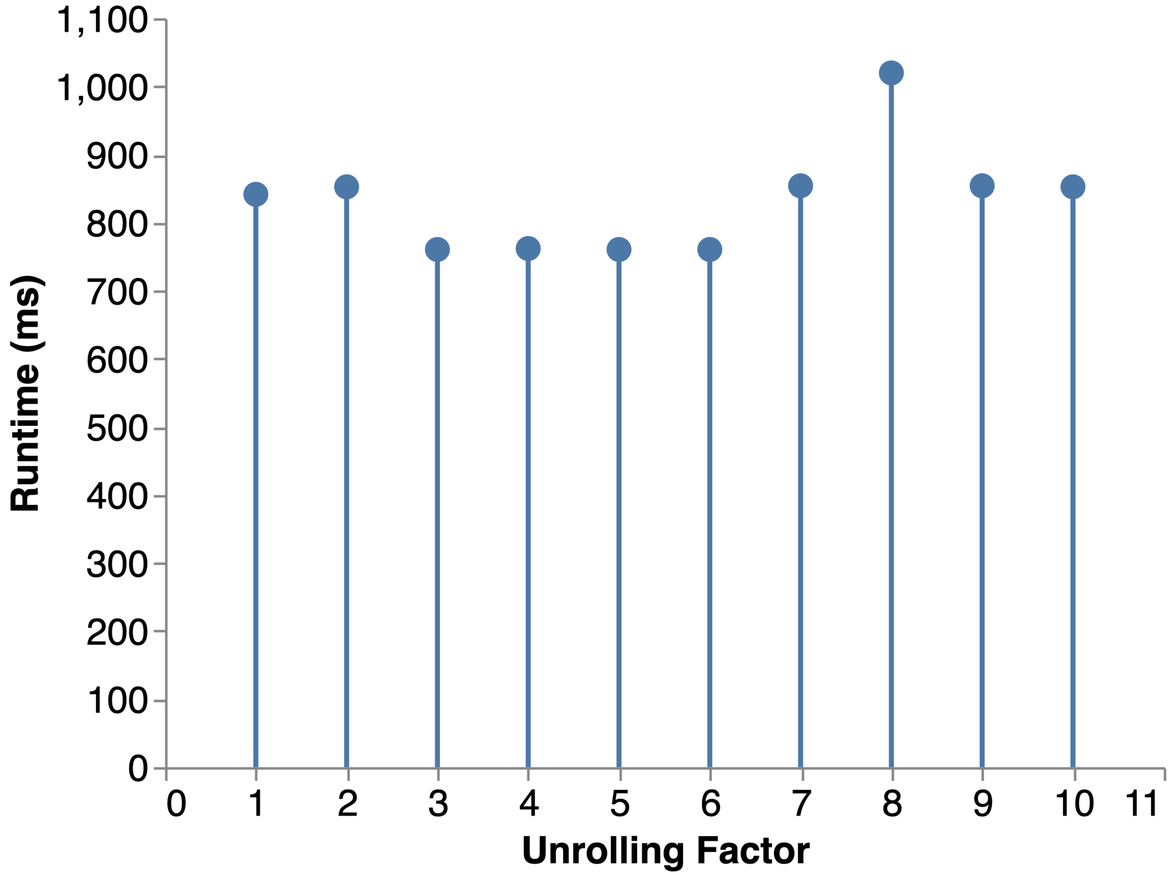
Multiplexers

Hardware

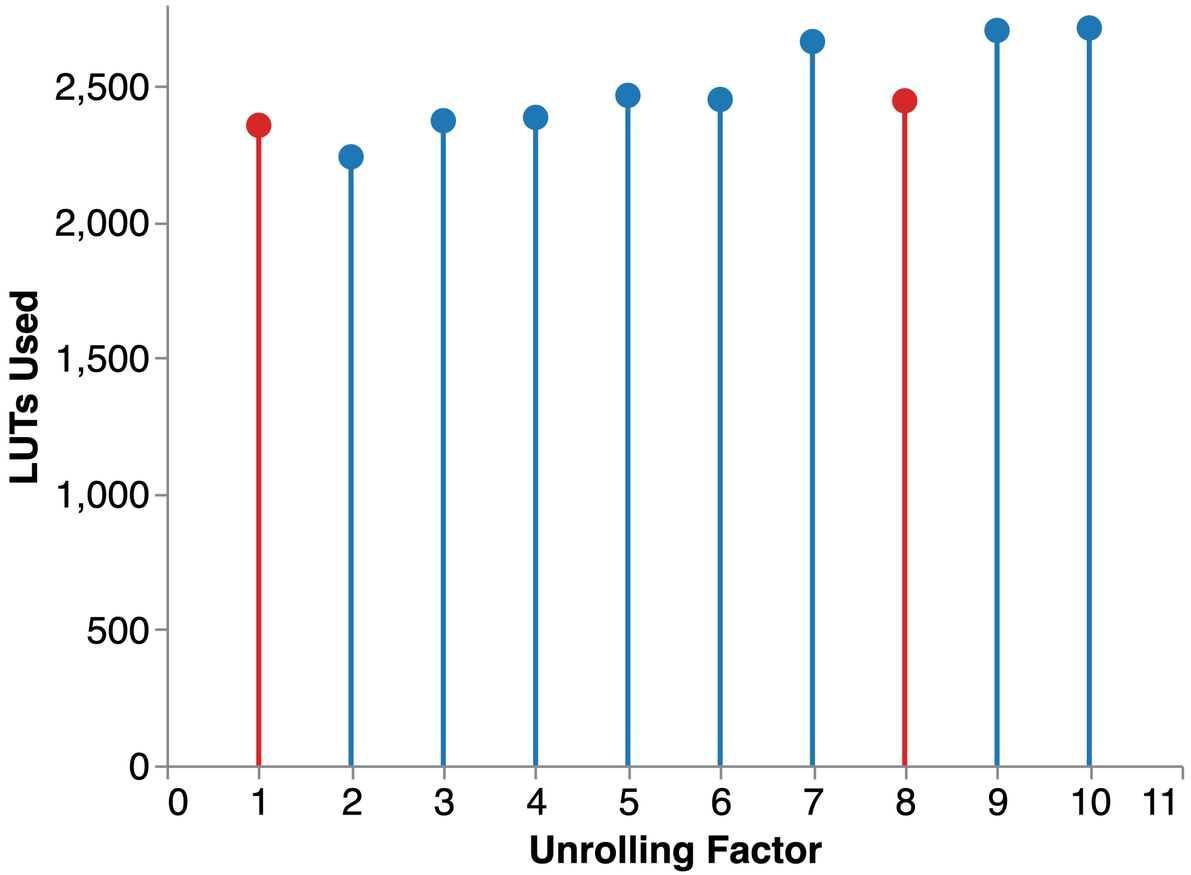


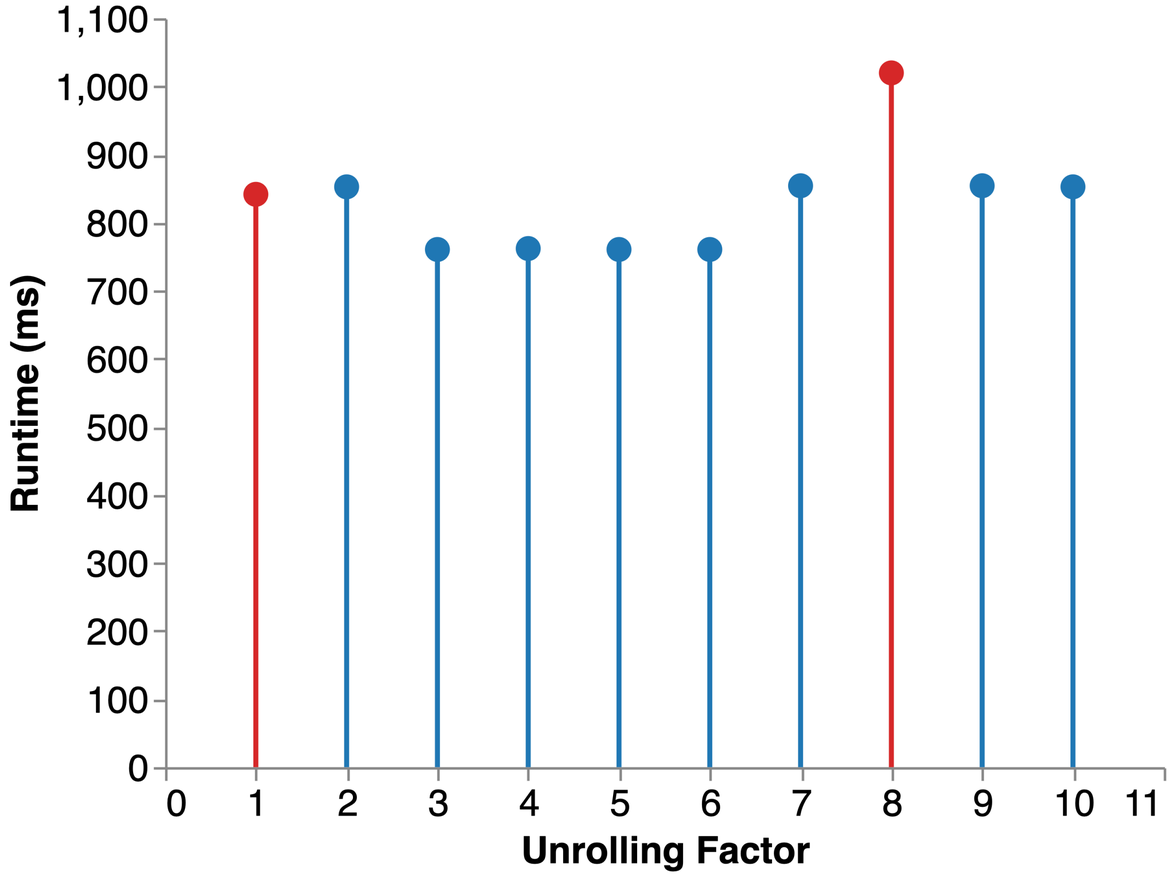
Multiplexers





Area-Latency trade-offs





Area-Latency trade-offs

Un·pre·dict·a·ble

adjective

Behaving in a way that is not easily predicted.

2012.1 Vivado HLS - Mutually exclusive memory access is not implemented with MUX on addresses, and reports: "@W [SCHED-69] Unable to schedule 'load' operation on array 'x' due to limited resources (II = 1) "

pkuruganti
Observer

09-21-2017 02:38 PM

1,879 Views
Registered: 02-21-2013

SdAccle 2017.2 + KCU1500: [XOCC 204-69] Unable to schedule 'load/store' operation

tsoliman
Visitor

02-06-2019 04:47 AM - edited 02-06-2019 04:49 AM

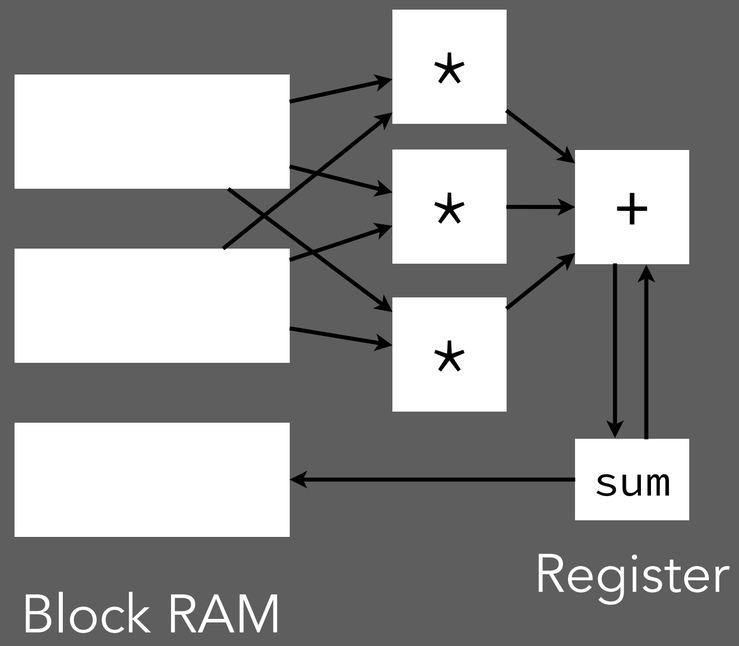
1,973 Views
Registered: 02-06-2019

Unable to schedule 'load' operation

Super secret™ accelerator

```
int m1[512][512];
#pragma PARTITION factor=3
int m2[512][512];
#pragma PARTITION factor=3
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      #pragma HLS UNROLL factor=3
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

Hardware

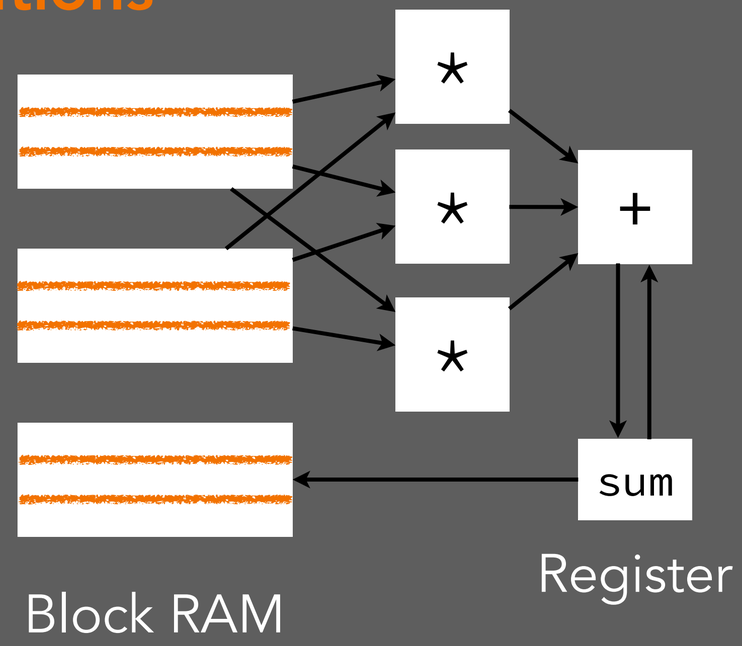


Super secret™ accelerator

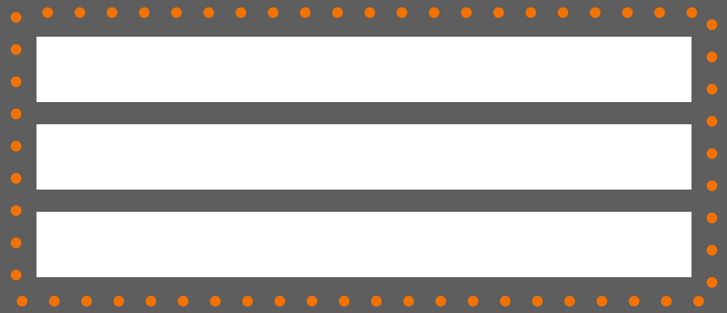
```
int m1[512][512];
#pragma PARTITION factor=3
int m2[512][512];
#pragma PARTITION factor=3
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      #pragma HLS UNROLL factor=3
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

Hardware

Memory Partitions

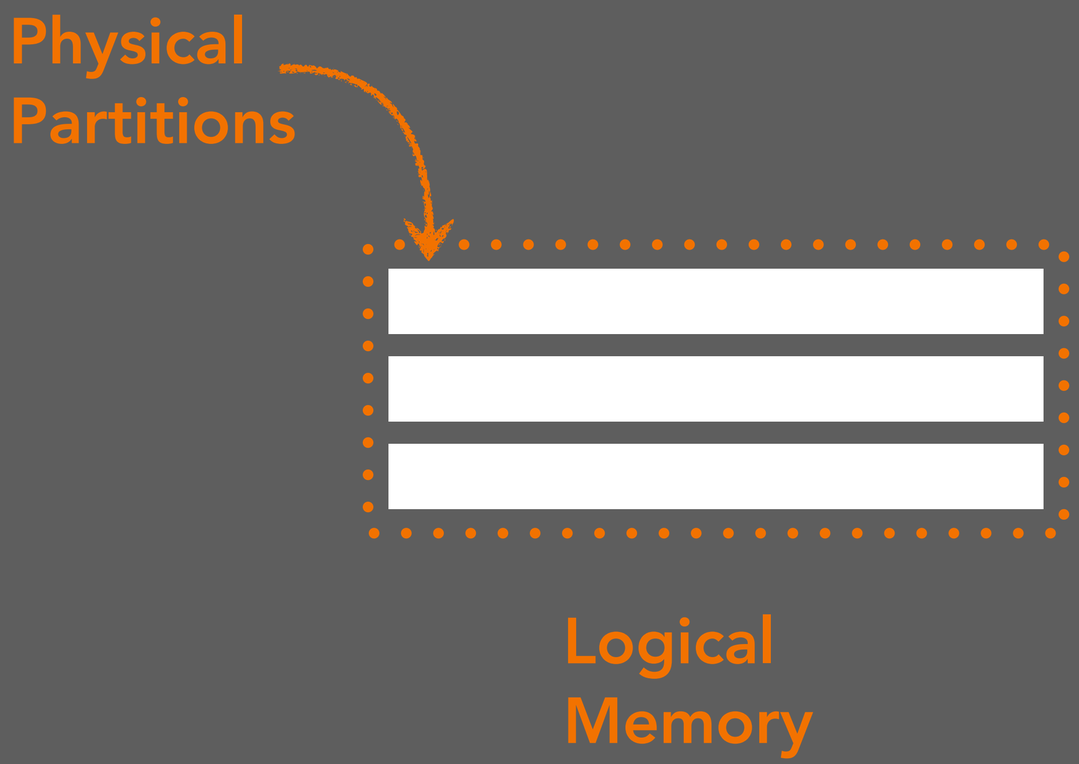


Hardware



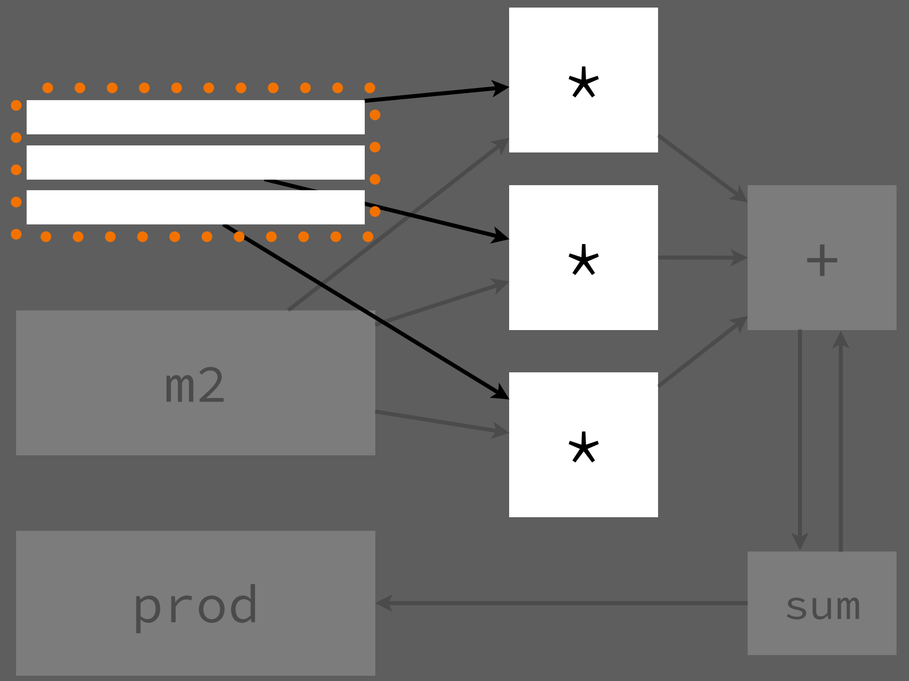
Logical
Memory

Hardware



Generated hardware

Memory Partitions

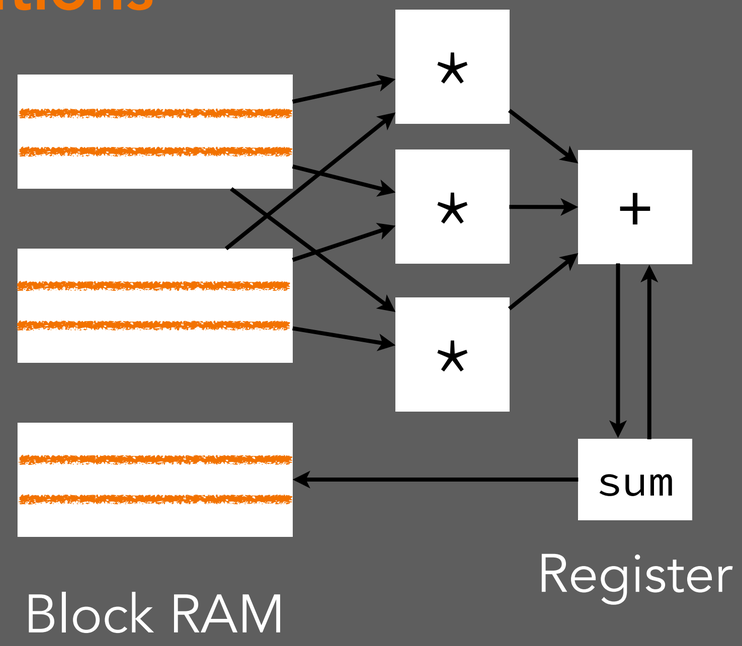


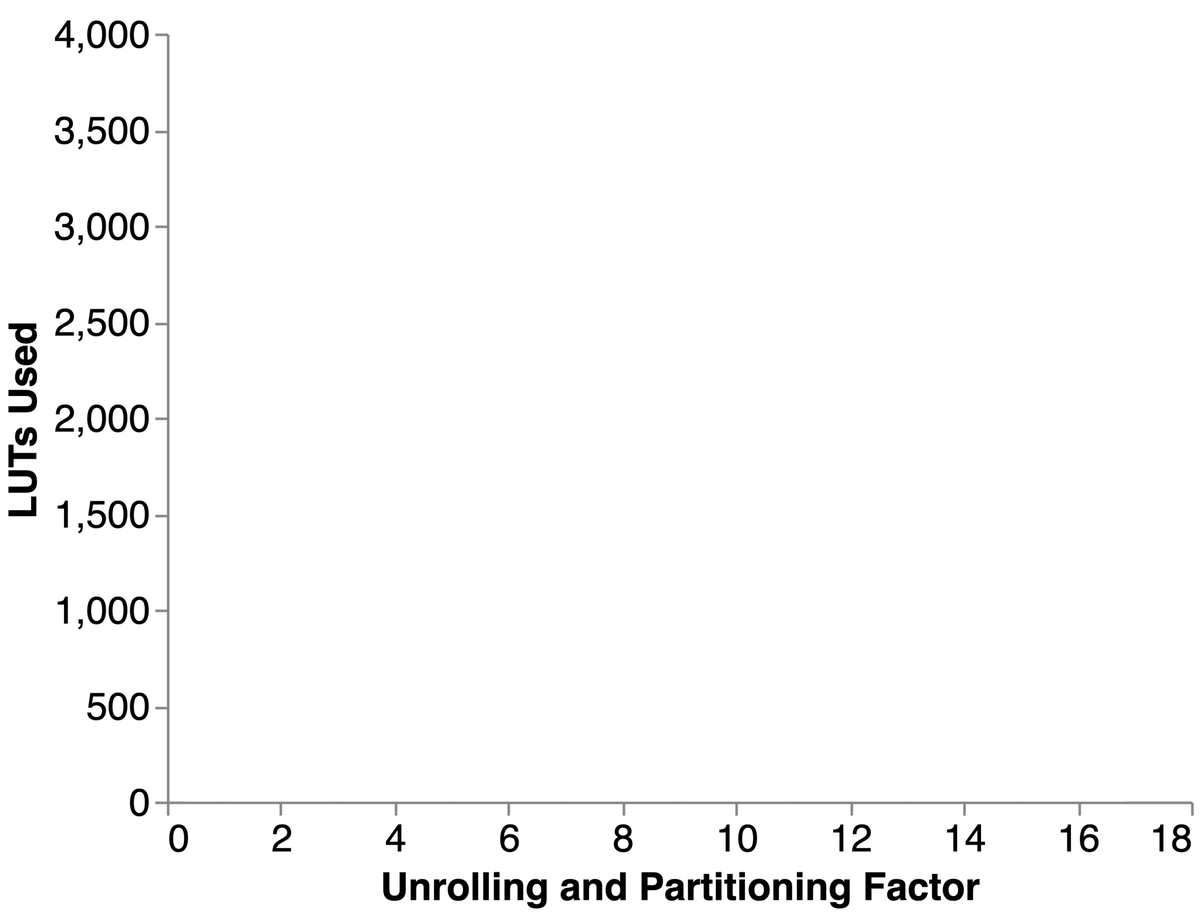
Super secret™ accelerator

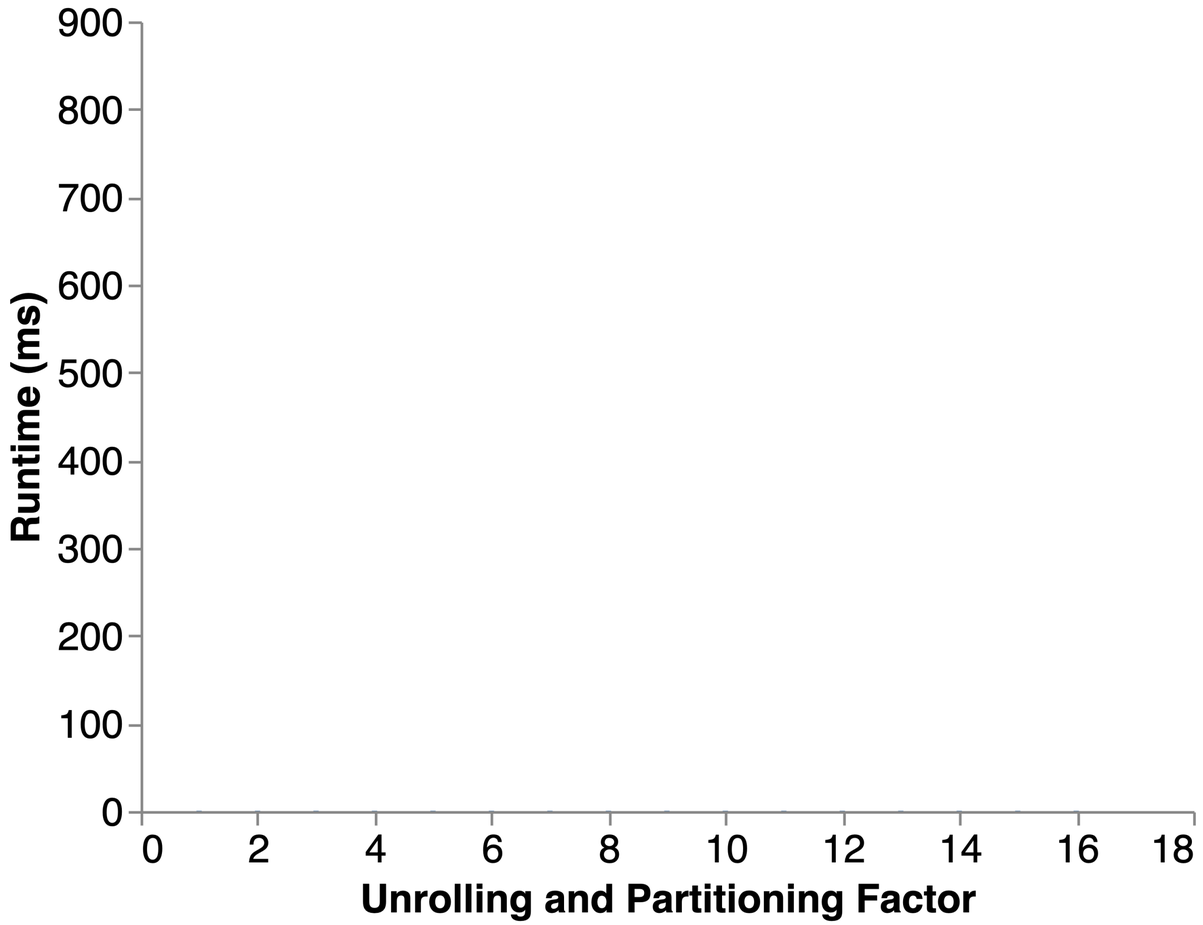
```
int m1[512][512];
#pragma PARTITION factor=3
int m2[512][512];
#pragma PARTITION factor=3
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      #pragma HLS UNROLL factor=3
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

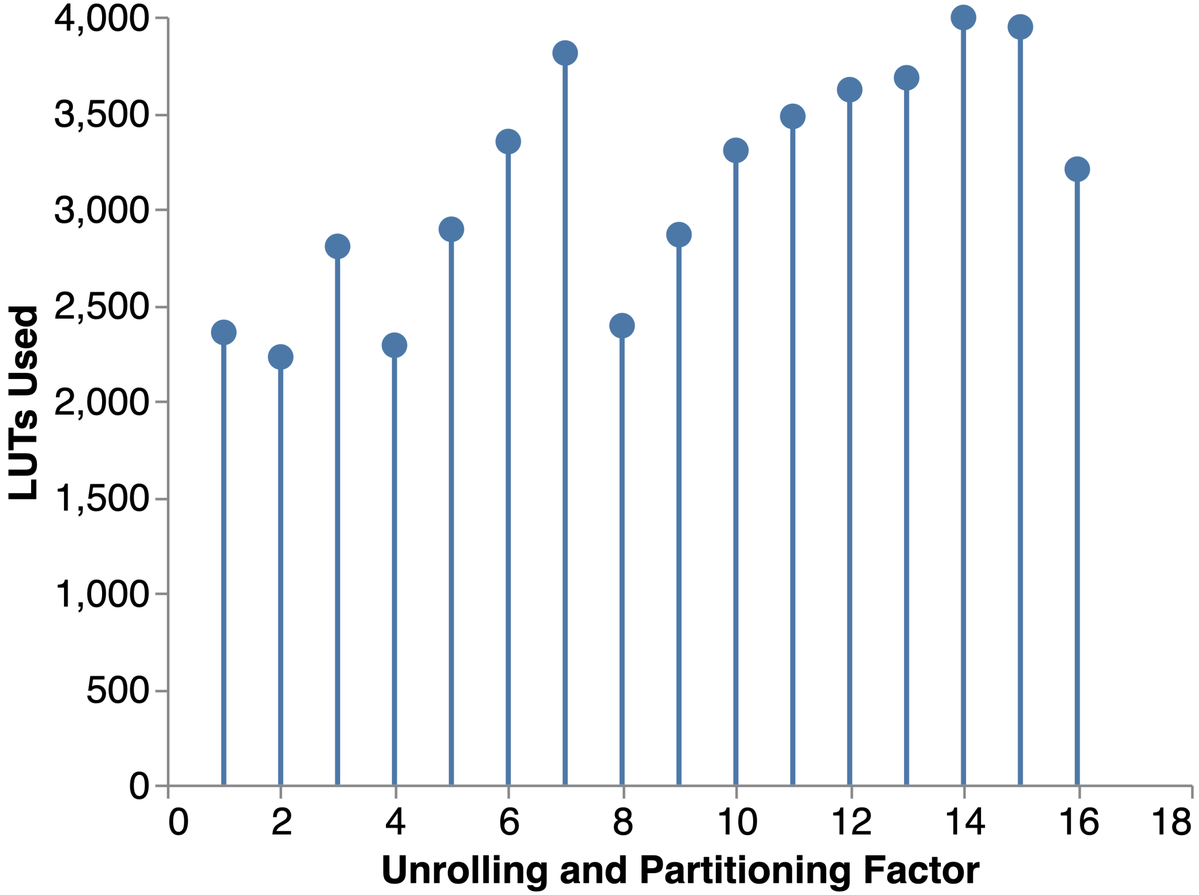
Hardware

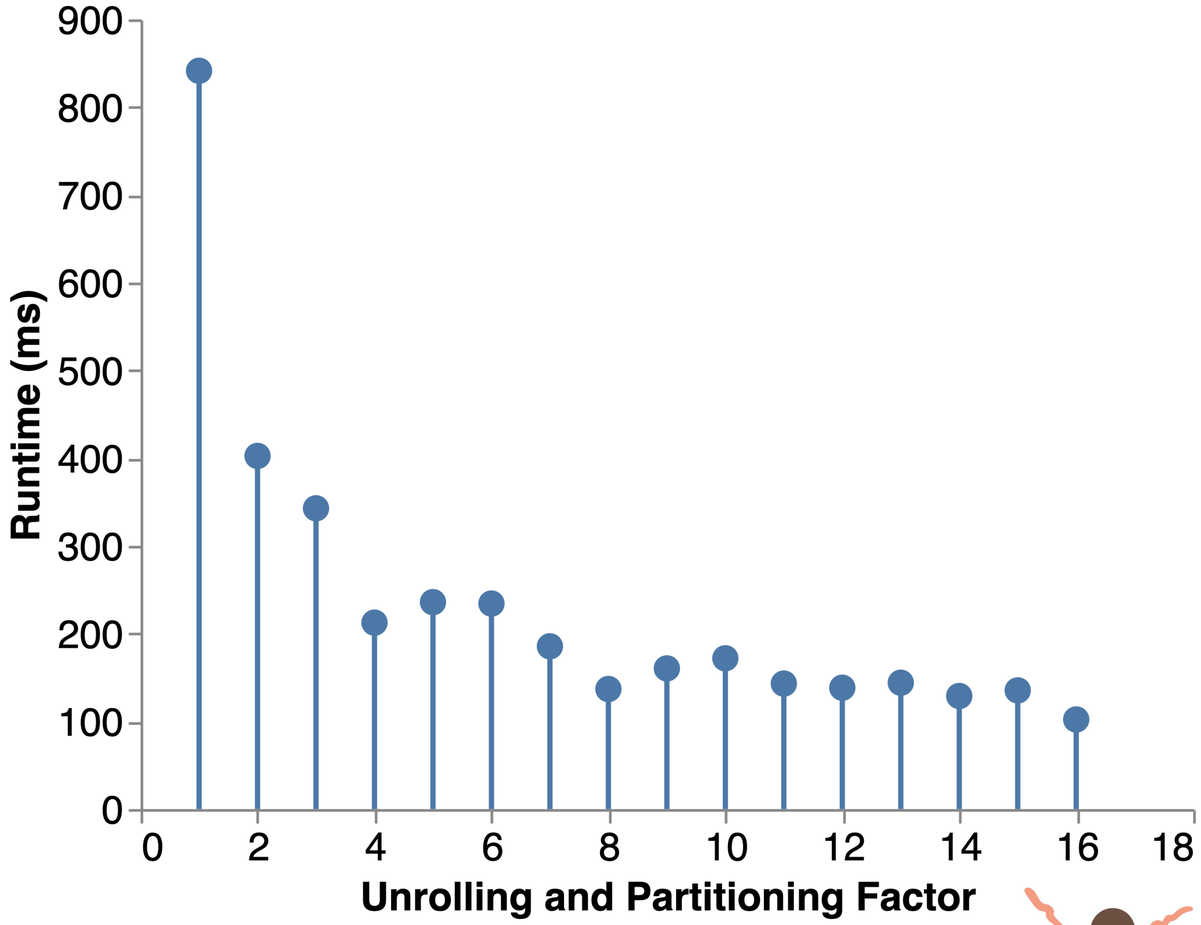
Memory Partitions

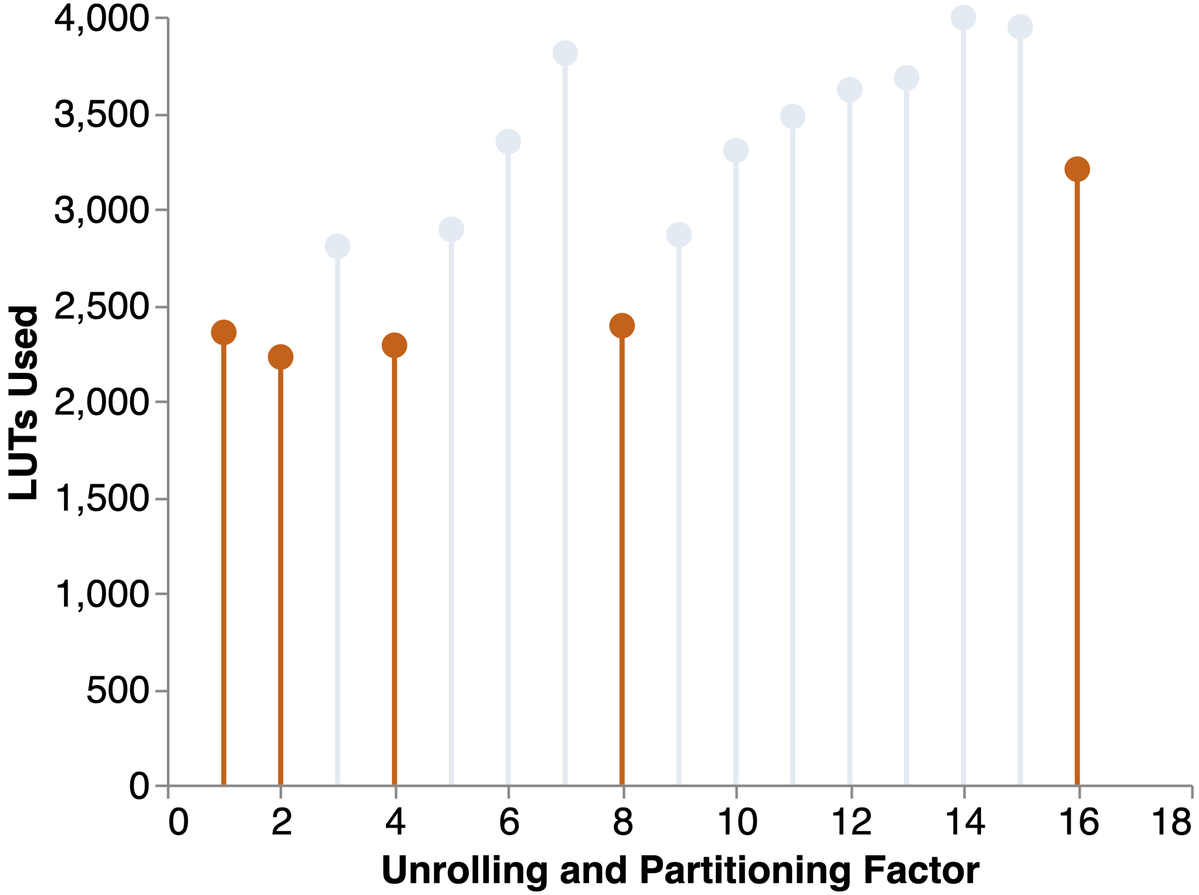


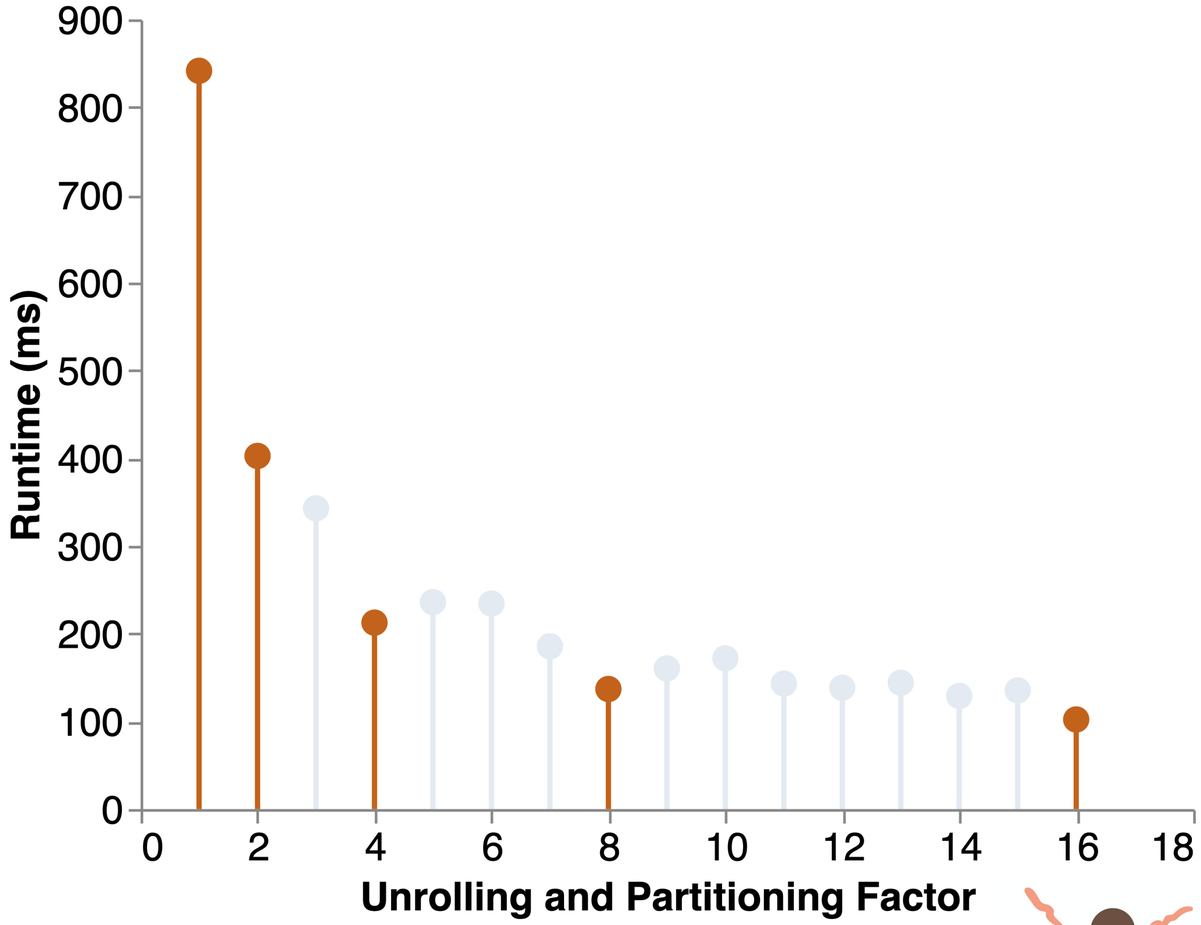

LUTs Used
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor


Runtime (ms)
900
800
700
600
500
400
300
200
100
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor


LUTs Used
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor


Runtime (ms)
900
800
700
600
500
400
300
200
100
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor


LUTs Used
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor

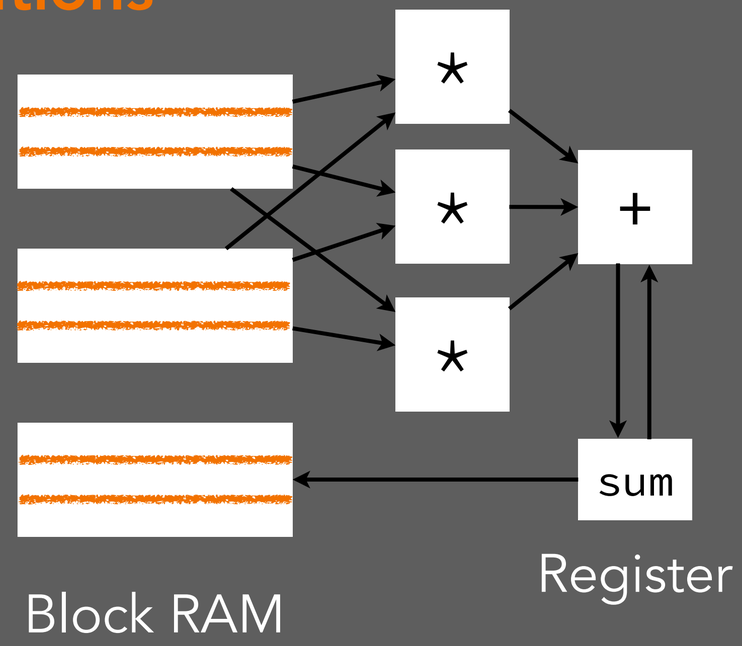

Runtime (ms)
900
800
700
600
500
400
300
200
100
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor

Super secret™ accelerator

```
int m1[512][512];
#pragma PARTITION factor=3
int m2[512][512];
#pragma PARTITION factor=3
int prod[512][512];
for (int i = 0; i < 512; i++) {
  for (int j = 0; j < 512; j++) {
    int sum = 0;
    for (int k = 0; k < 512; k++) {
      #pragma HLS UNROLL factor=3
      sum += m1[i][k] * m2[k][j];
    }
    prod[i][j] = sum;
  }
}
```

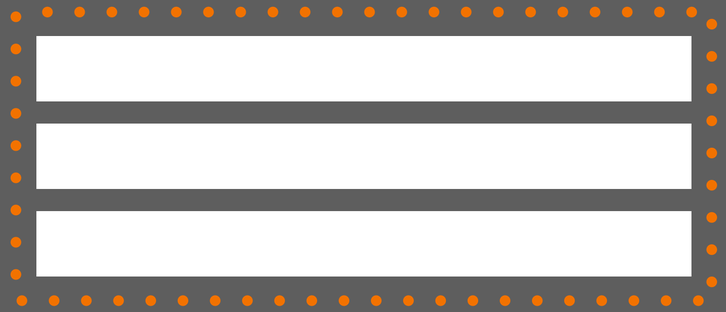
Hardware

Memory Partitions



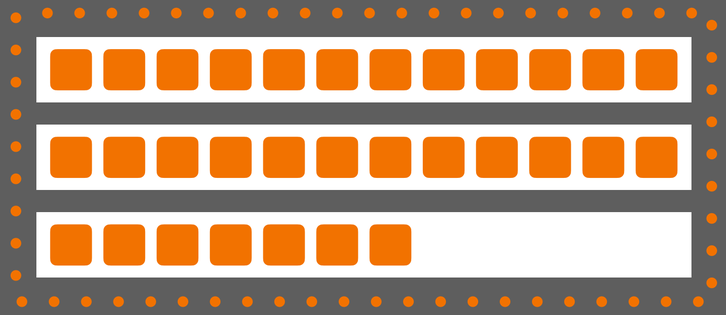
Hardware

$$512 \% 3 \neq 0$$



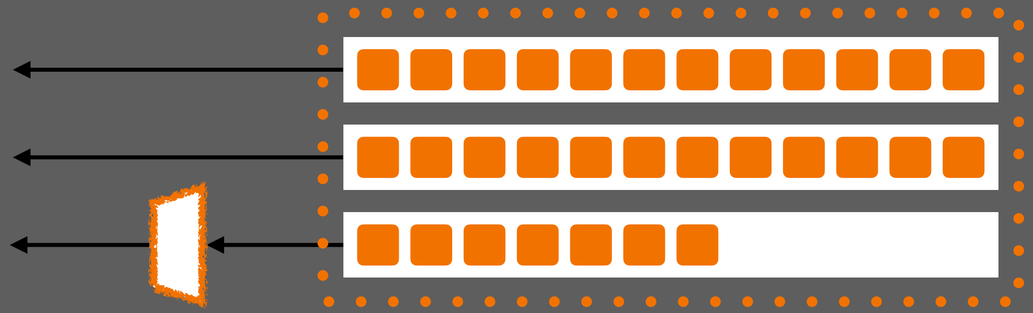
Hardware

$$512 \;\%\; 3 \neq 0$$

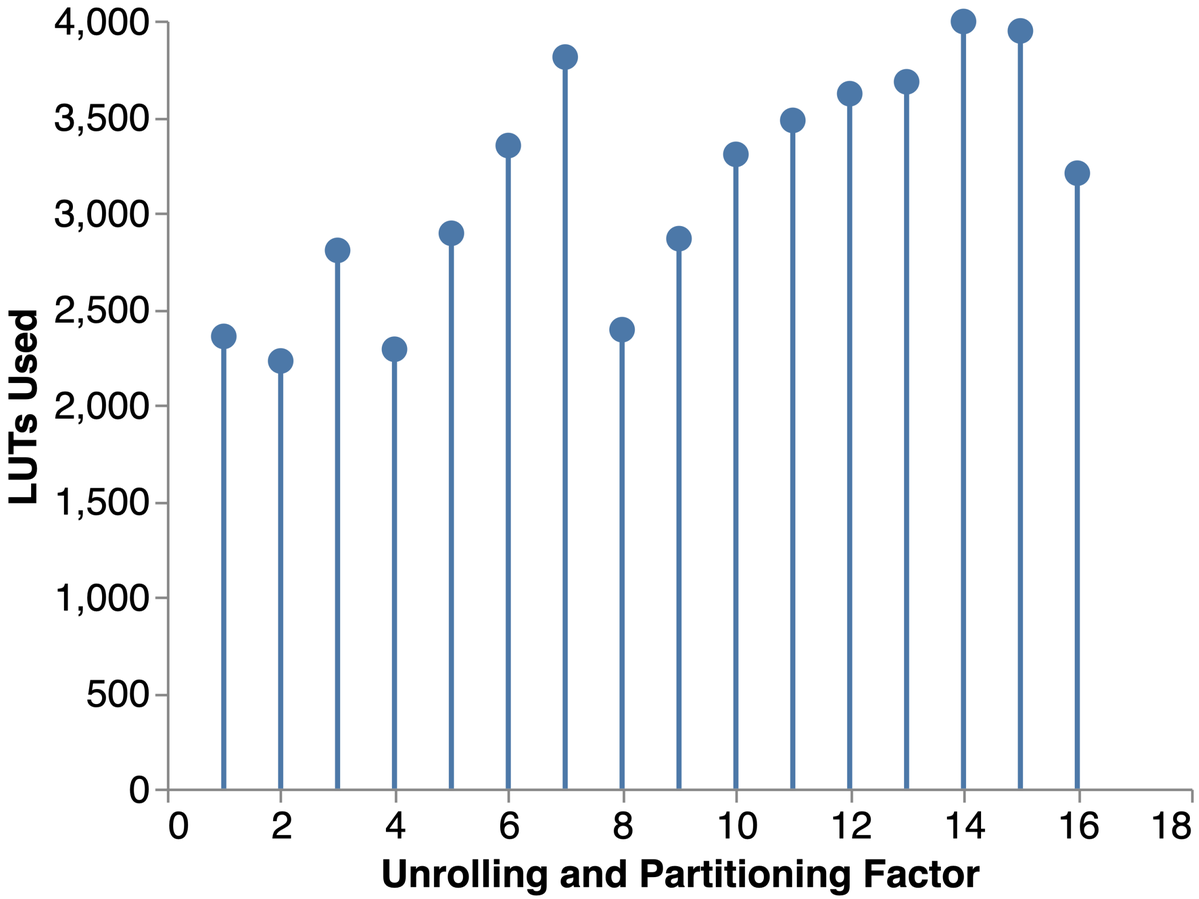


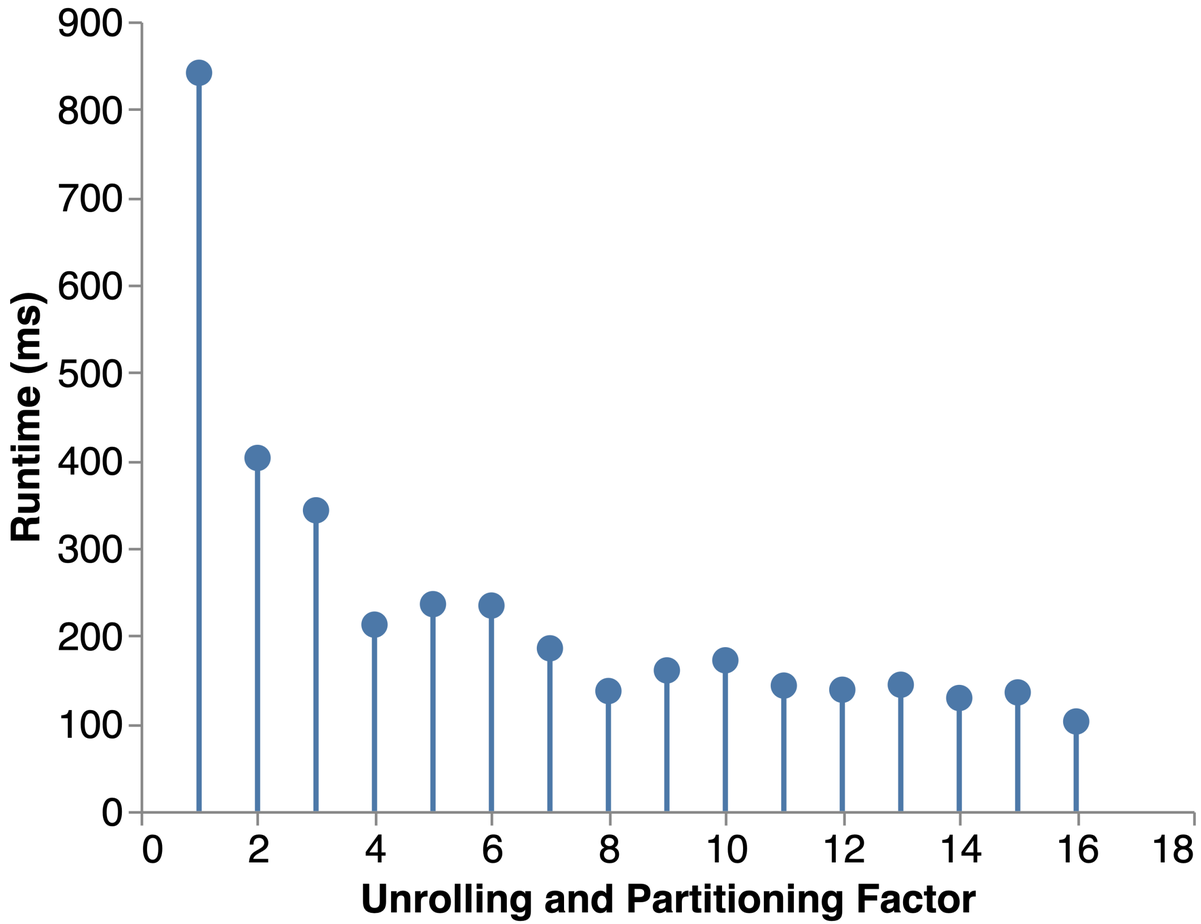
Mismatched
Partition Sizes

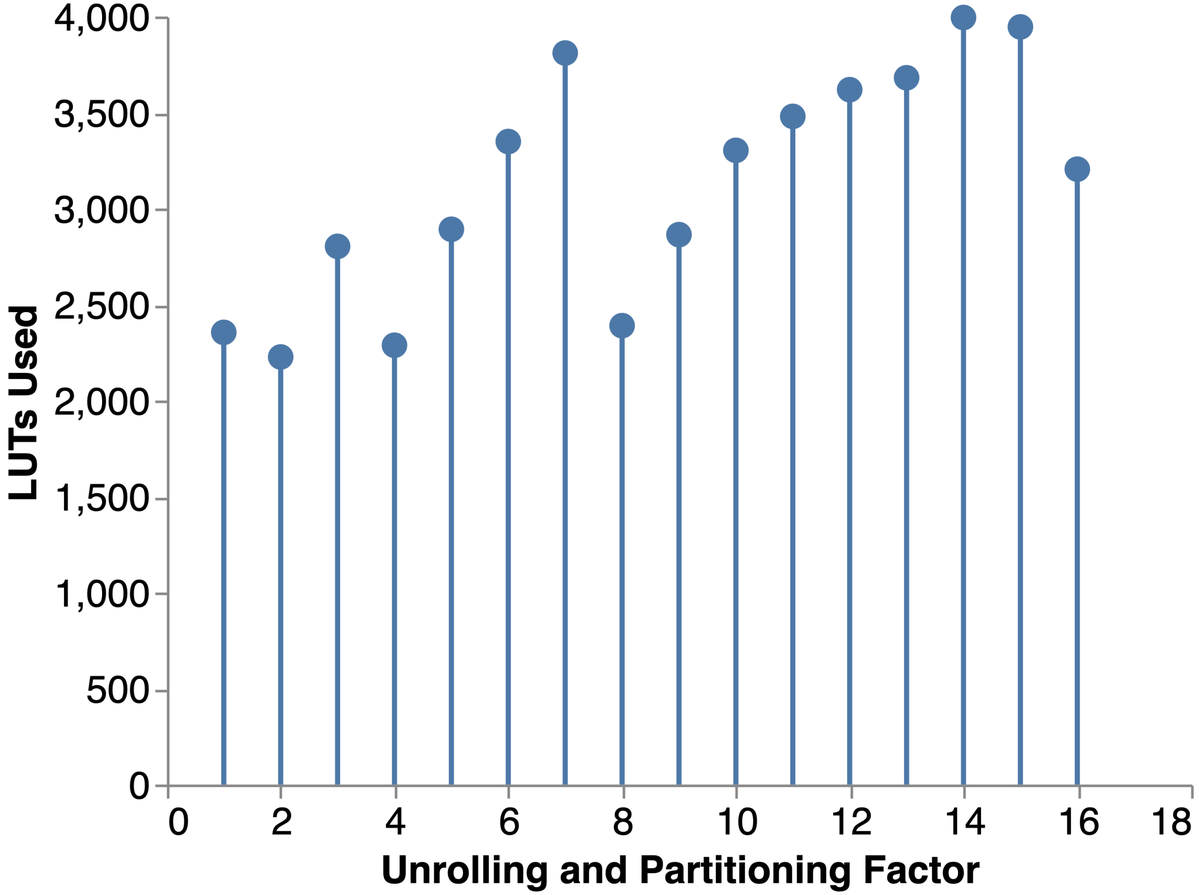
Hardware

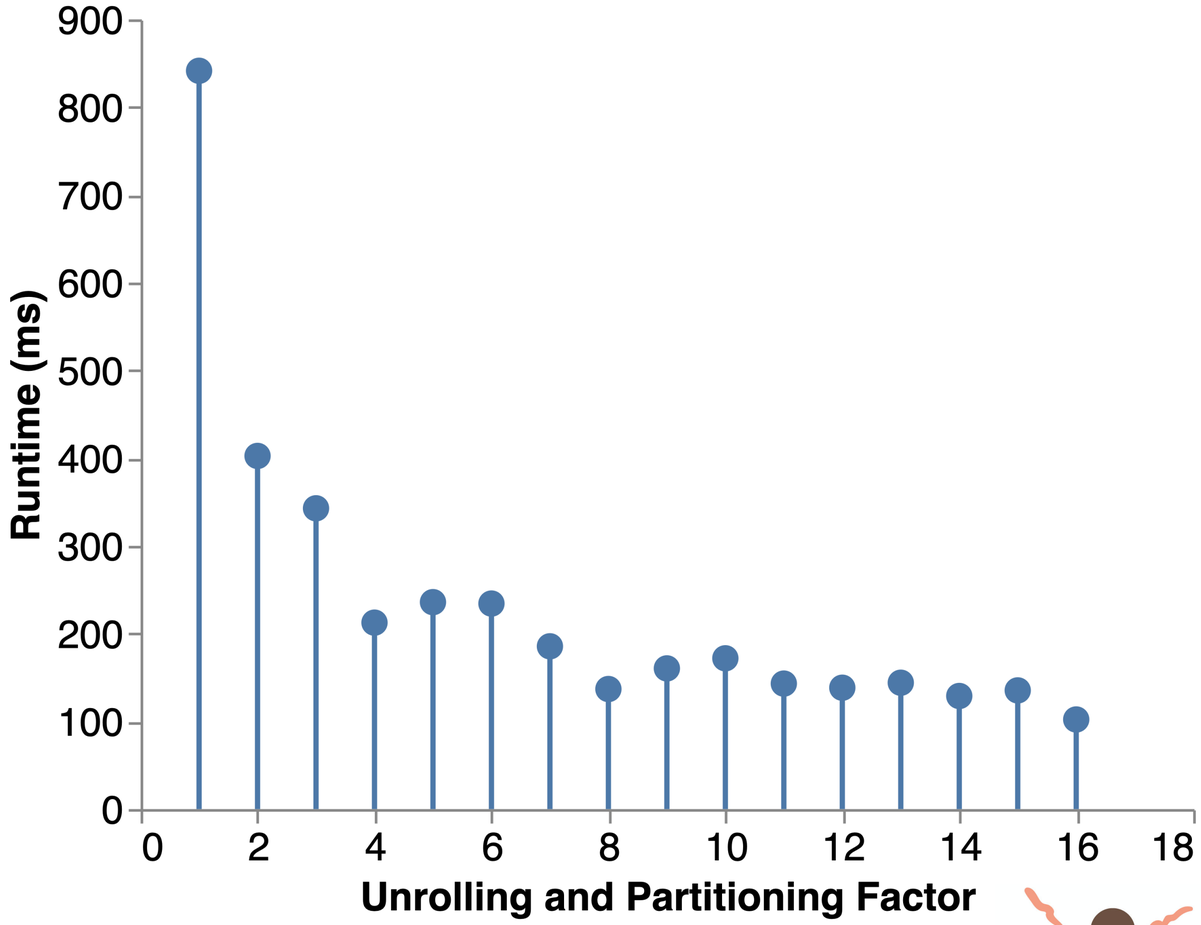


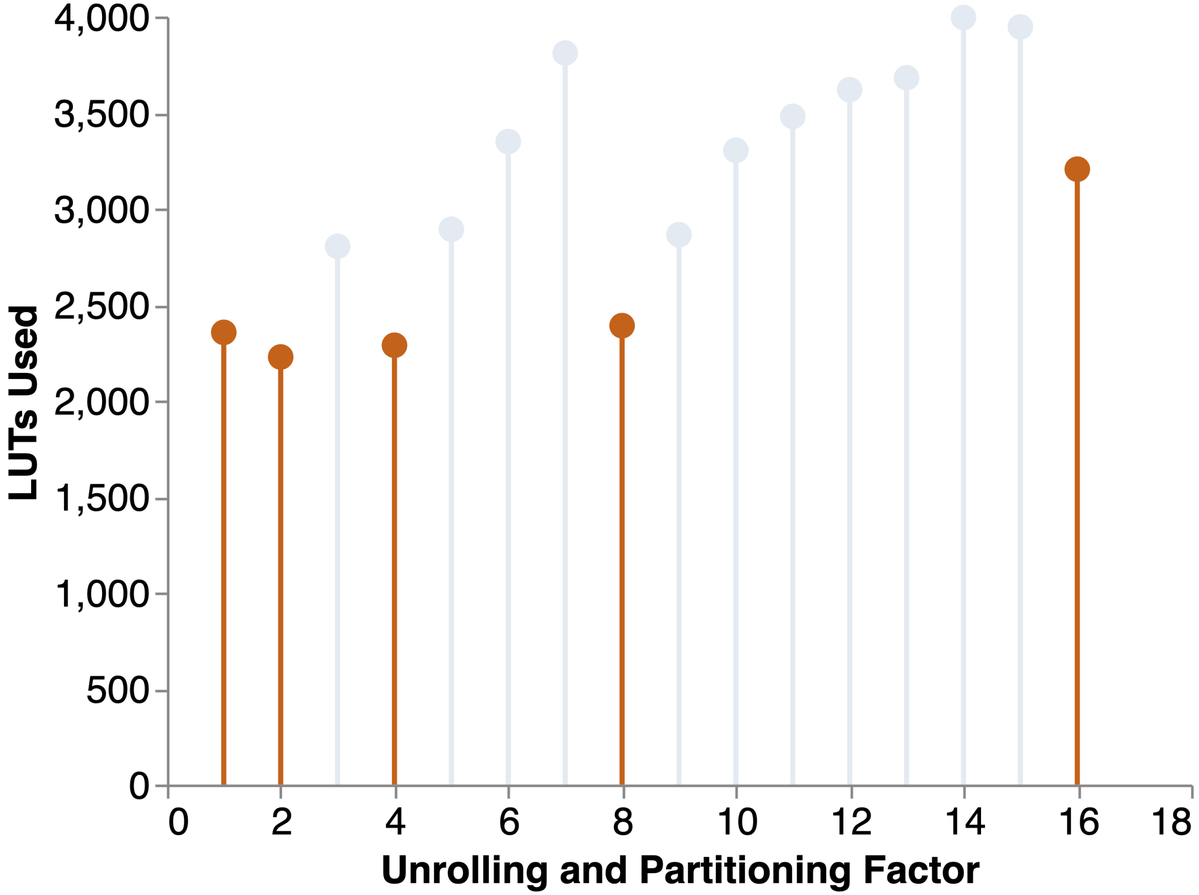
Mismatched
Partition Sizes

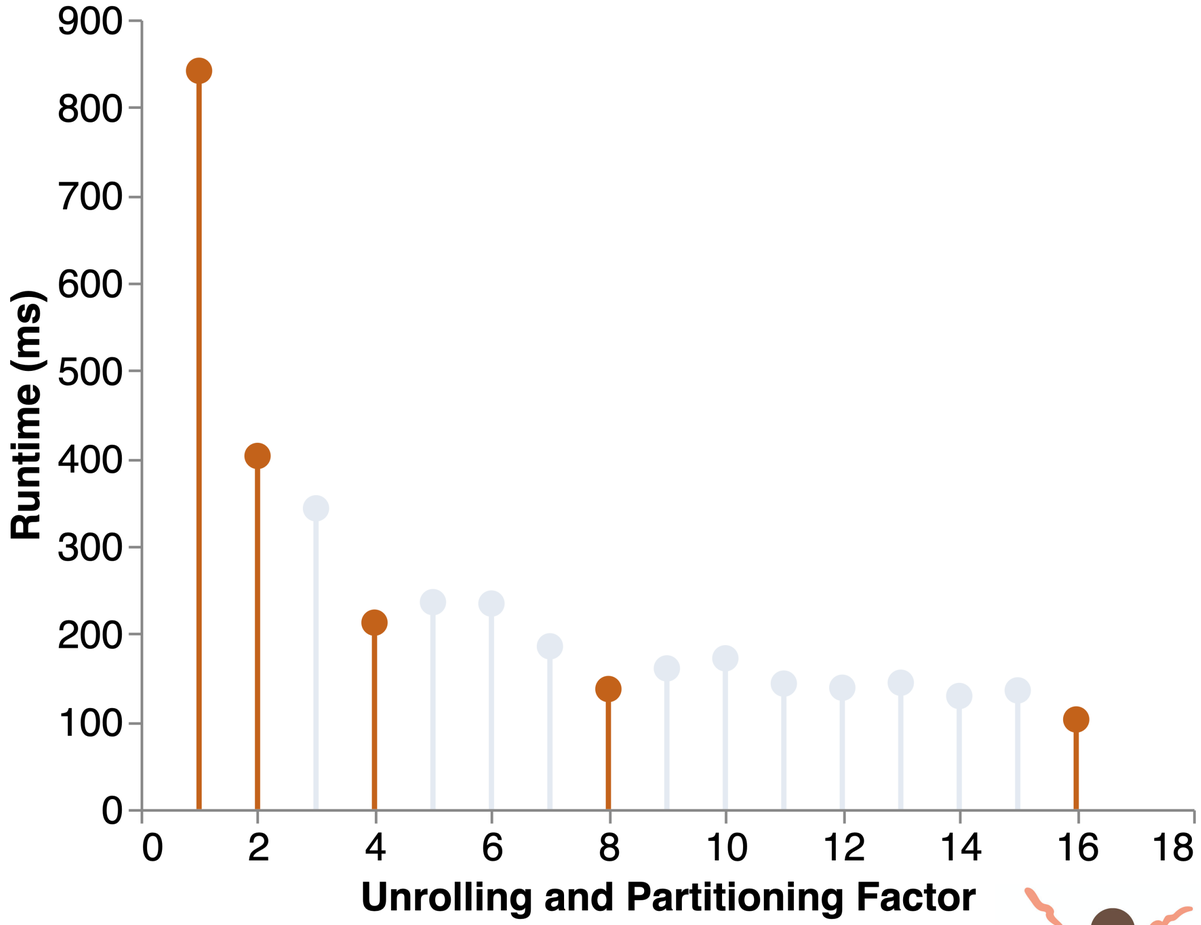

LUTs Used
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor


Runtime (ms)
900
800
700
600
500
400
300
200
100
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor


LUTs Used
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor


Runtime (ms)
900
800
700
600
500
400
300
200
100
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor


LUTs Used
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor


Runtime (ms)
900
800
700
600
500
400
300
200
100
0
0
2
4
6
8
10
12
14
16
18
Unrolling and Partitioning Factor

HLS Really Works!

HLS Really Works!******

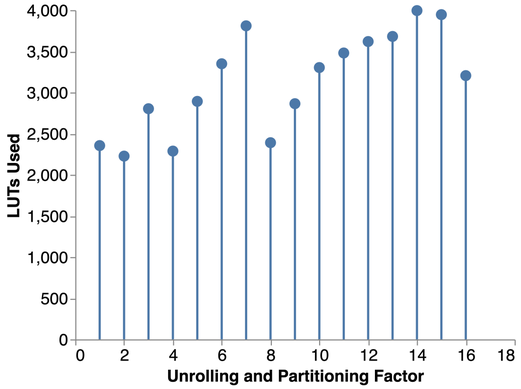
* when you unroll designs
* when unrolling and partitioning are aligned
* when partitioning and memory sizes are aligned
* when ports times partitioning is a factor of unrolling
* when memory accesses are easily analyzable
* when reduction patterns are easily analyzable

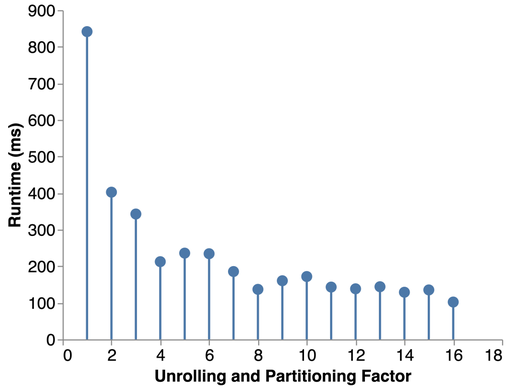
HLS Really Works!******

HLS Really Works!******

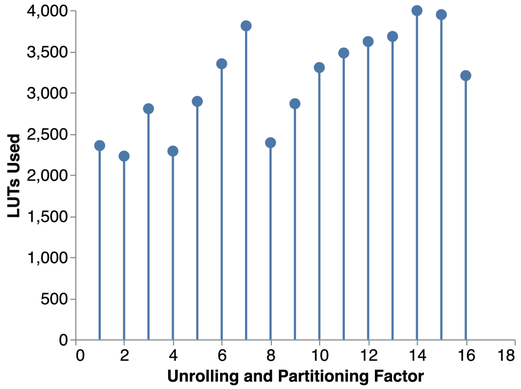
The language doesn’t capture
timing and **resource constraints**.

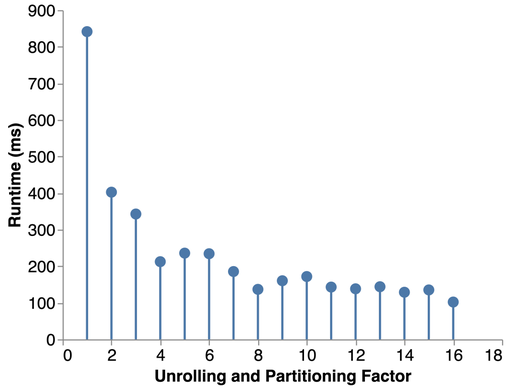
Unpredictable





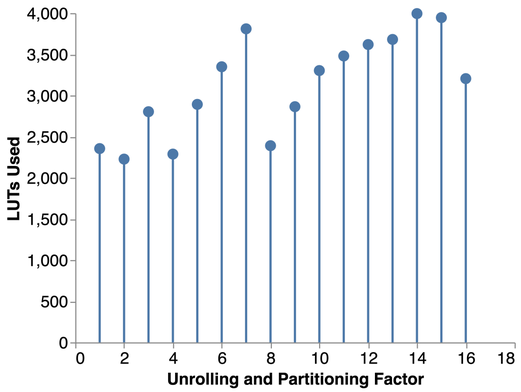
Unpredictable

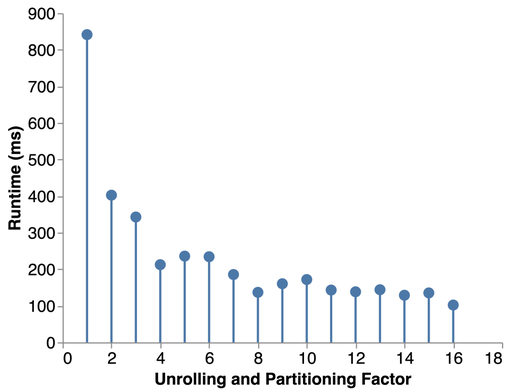




16 designs =
32 hours of
compilation

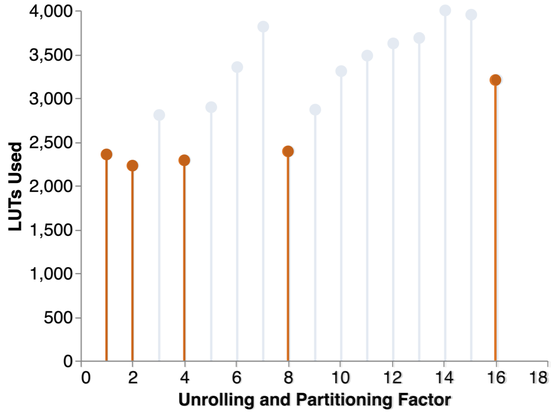
Unpredictable

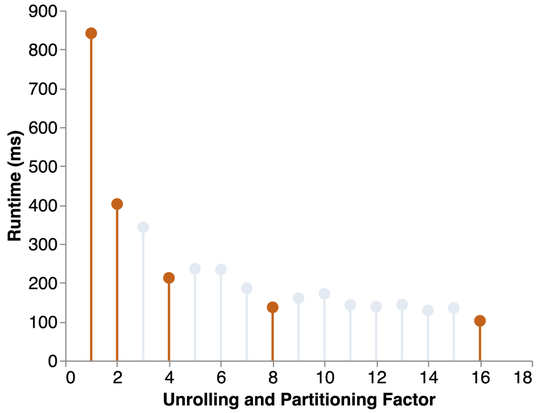




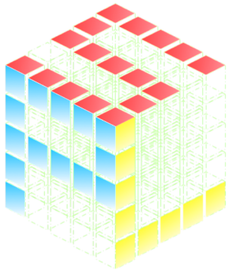
16 designs =
32 hours of
compilation

Predictable

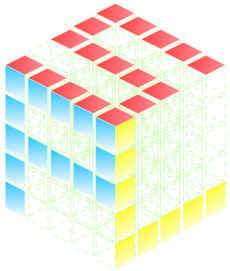




4 designs =
8 hours of
compilation



Spatial: A Language and Compiler for Application Accelerators

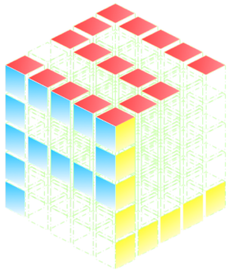


Spatial: A Language and Compiler for Application Accelerators

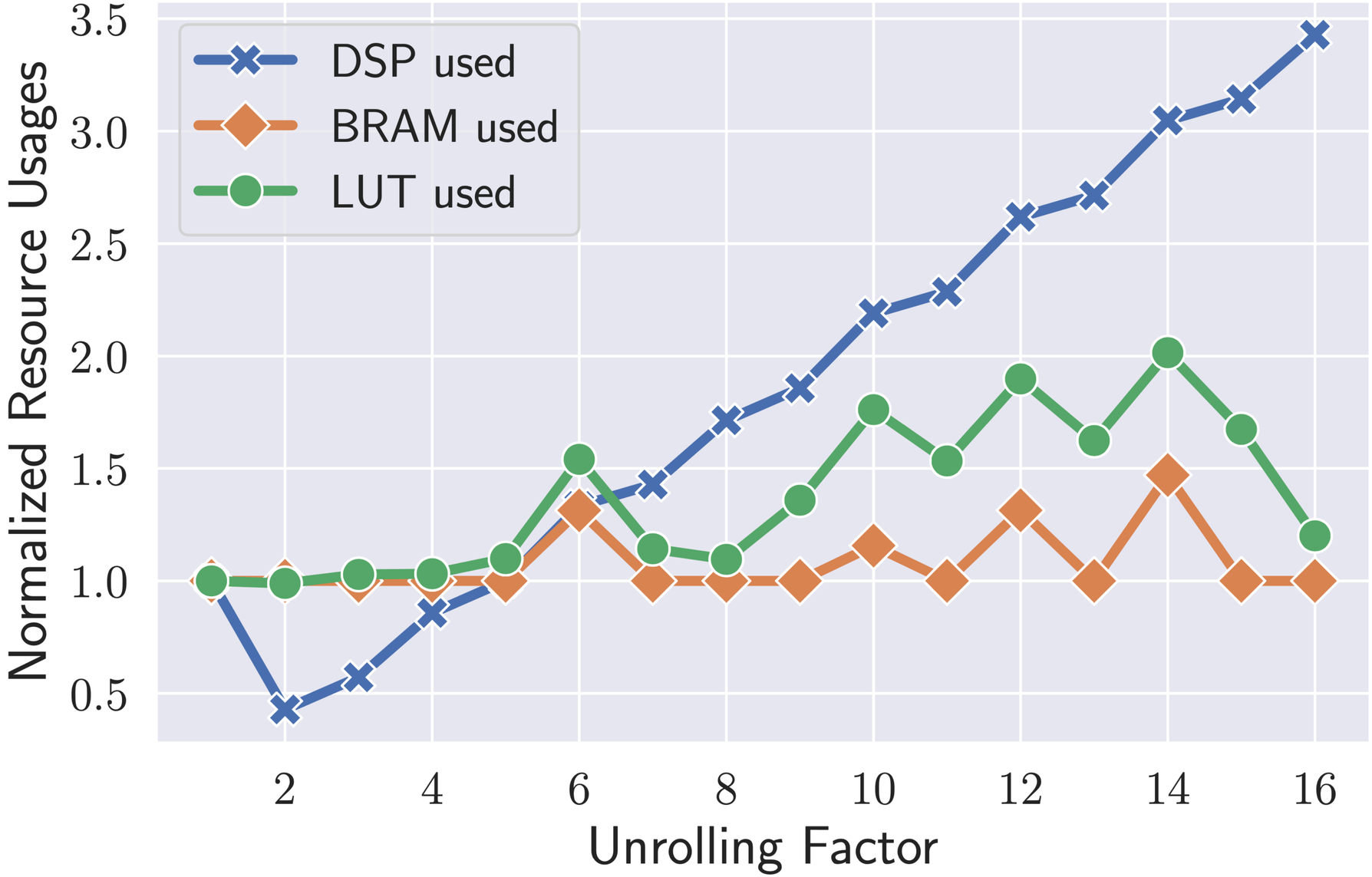
```
1  function GroupAccesses:
2    input: A → set of reads or writes to m
3    G = ∅ set of sets of compatible accesses
4    for all accesses a in A:
5      for all sets of accesses g in G:
6        if IComp(a, a') for all a' in g then
7          add a to g
8          break
9        else add {a} to G
10   return G
11 end function
12
13 function ConfigureMemory:
14   input: A_r → set of reads of m
15   input: A_w → set of writes to m
16   G_r = GroupAccesses(A_r)
17   G_w = GroupAccesses(A_w)
18   I = ∅ set of memory instances
19   for all read sets R in G_r:
20     I_r = {R}
21     I_w = ReachingWrites(G_w, I_r)
22     i = BankAndBuffer(I_r, I_w)
23     for each inst in I:
24       I'_r = ReadSets[inst] + R
25       I'_w = ReachingWrites(G_w, I'_r)
26       if OComp(A_1, A_2) ∀A_1 ≠ A_2 ∈ (G_w ∪ I'_r) then:
27         i' = BankAndBuffer(I'_r, I'_w)
28         if Cost(i') < Cost(i) + Cost(inst) then:
29           remove inst from I
30           add i' to I
31           break
32     if i has not been merged then add i to I
33   return I
34 end function
```

Figure 7. Banking and buffering algorithm for calculating instances of on-chip memory m.

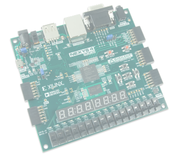
Automatic memory partitioning

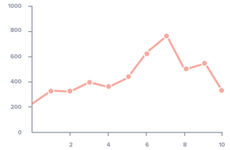


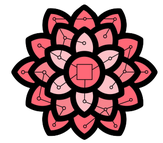
Spatial: A Language and Compiler for Application Accelerators



Ada's Journey

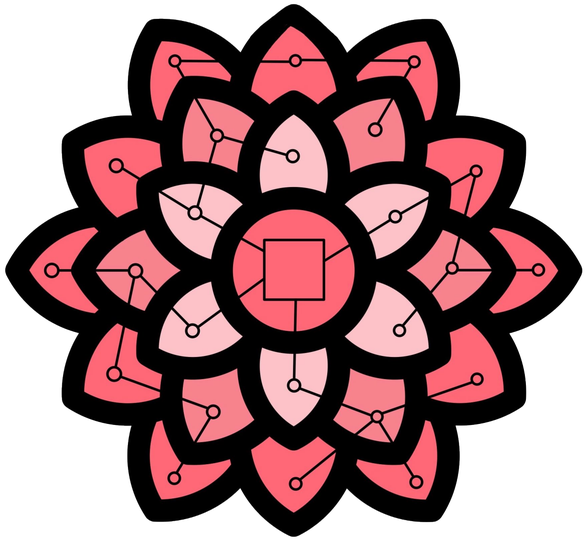


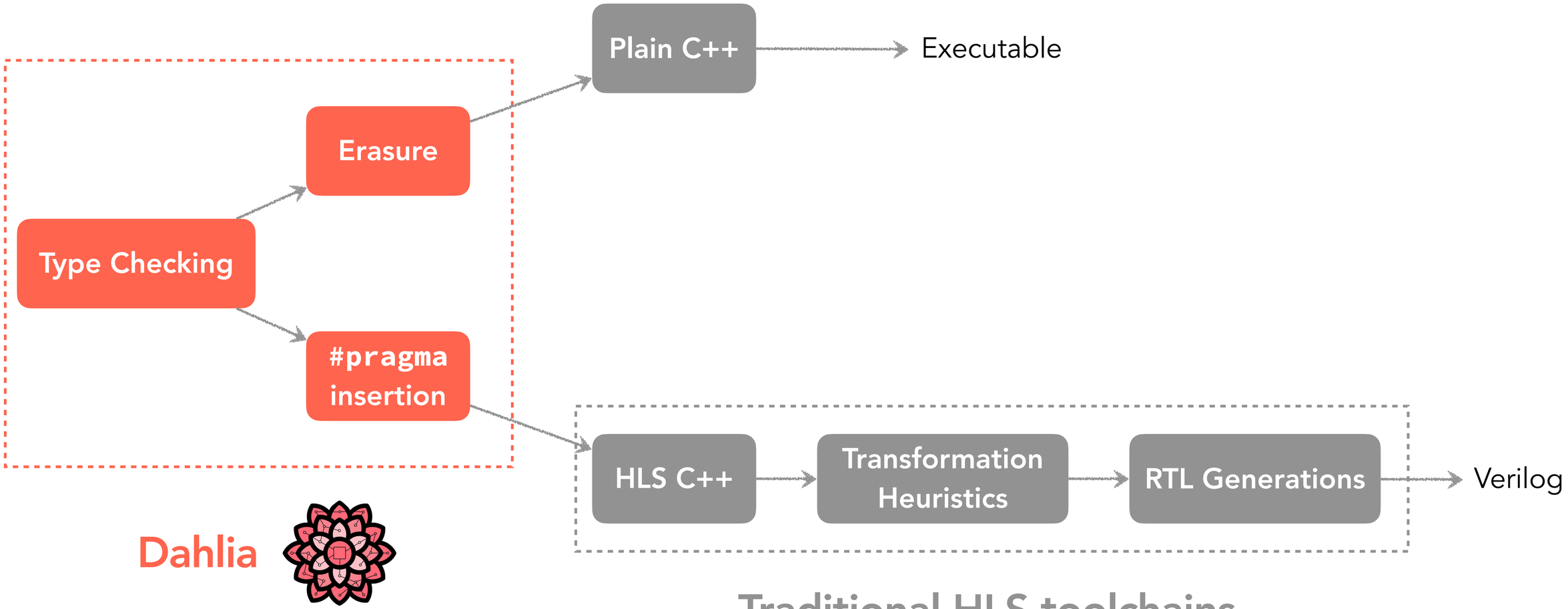




Dahlia

Predictable Accelerator Design




Plain C++
Executable
Erasure
Type Checking
#pragma
insertion
HLS C++
Transformation
Heuristics
RTL Generations
Verilog
Dahlia
Traditional HLS toolchains

Dahlia

Hardware

Dahlia

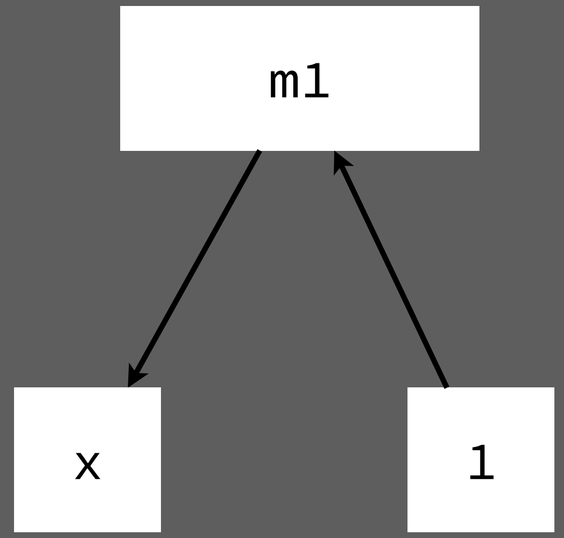
```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

Hardware

Dahlia

```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

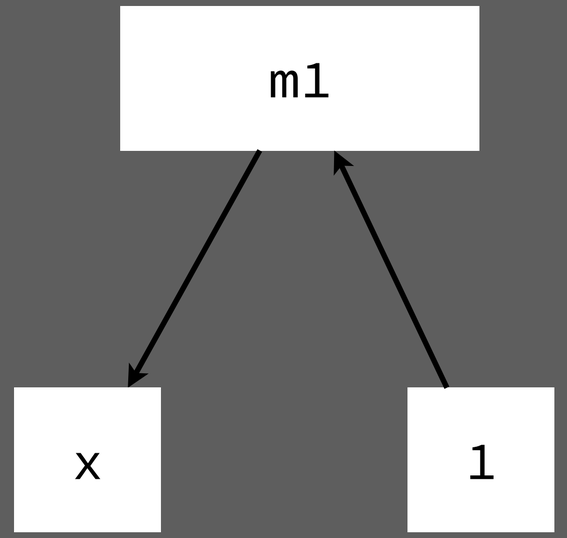
Hardware



Dahlia

```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

Hardware

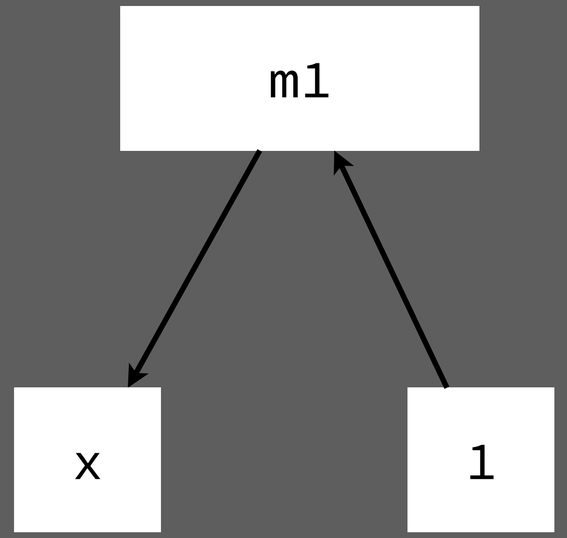


Dahlia

```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

Do these run concurrently?

Hardware

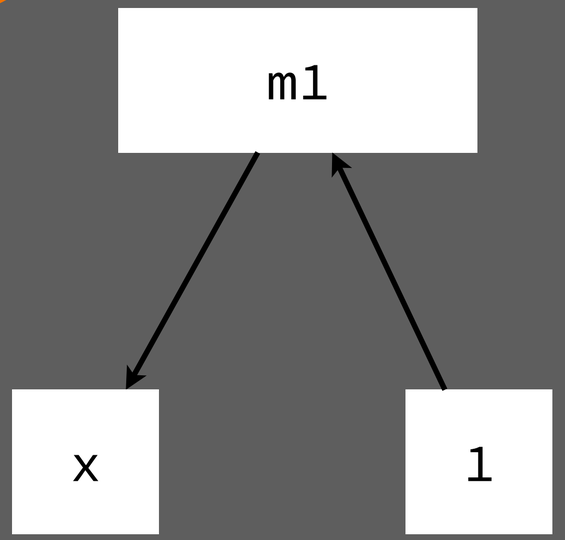


Dahlia

```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

Do these run concurrently?

Hardware



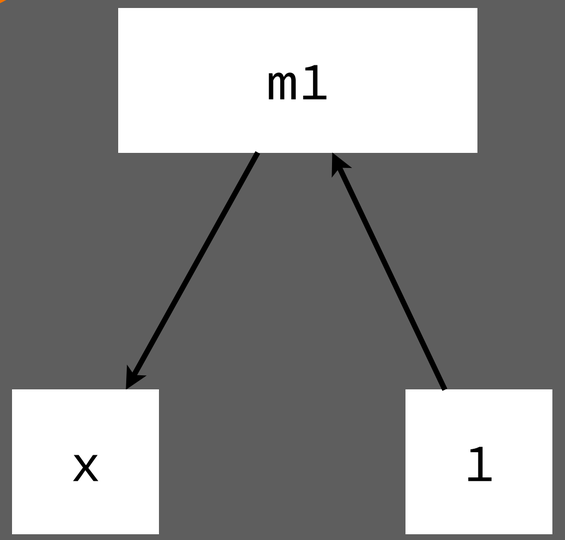
Dahlia

```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

Do these run concurrently?

Error: Affine resource ‘m1’ already used in this context.

Hardware



Dahlia

Affine Type System

```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

Dahlia

Affine Type System

```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

At most one **use** of a variable

Dahlia

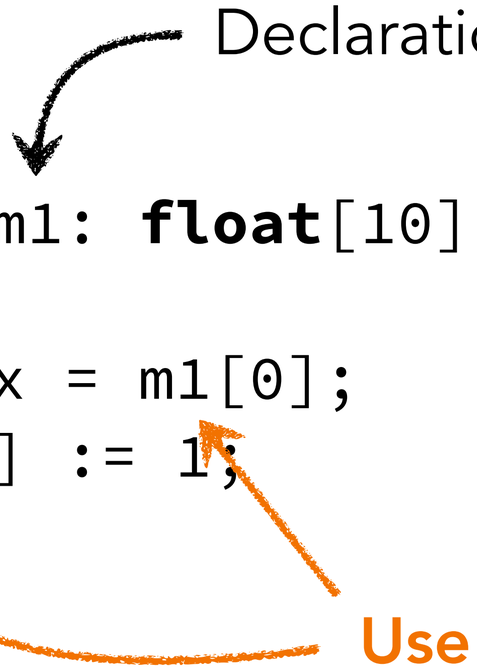
Affine Type System


```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

At most one **use** of a variable

Dahlia

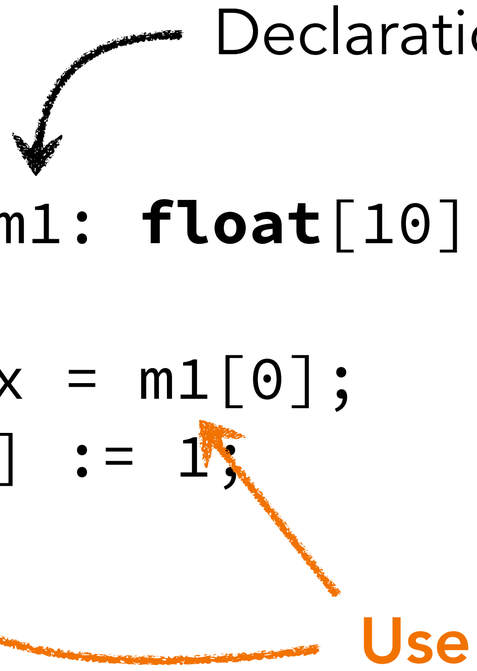
Affine Type System



At most one **use** of a variable

Dahlia

Affine Type System



At most one **use** of a variable

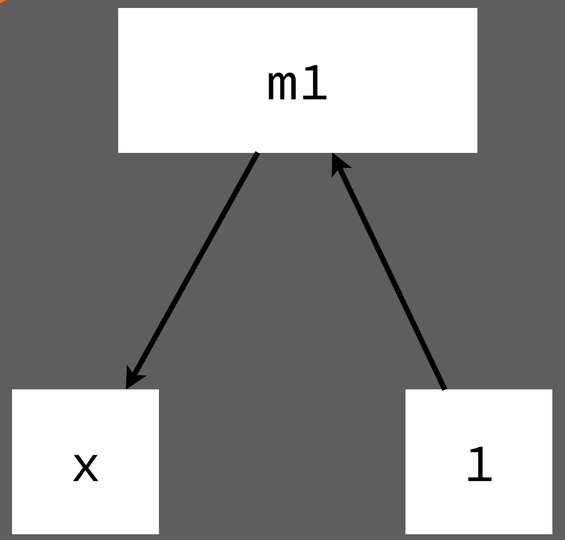
Error: Affine resource ‘m1’ already used in this context.

Dahlia

```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

Do these run concurrently?

Hardware



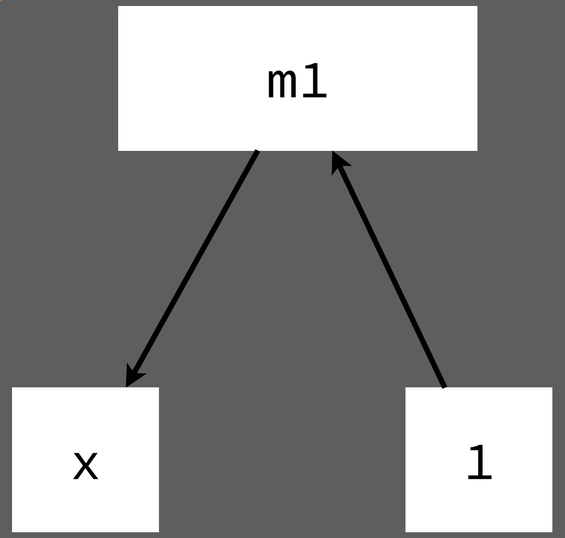
Dahlia

```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

Do these run concurrently?

Unordered Composition

Hardware



Dahlia

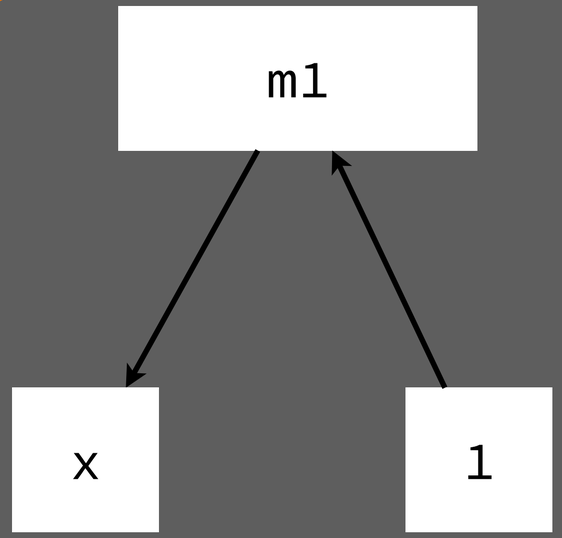
```
let m1: float[10];

let x = m1[0];
m1[1] := 1;
```

Do these run concurrently?

Unordered Composition

- Run in parallel respecting data flow.
- Consume affine resources.

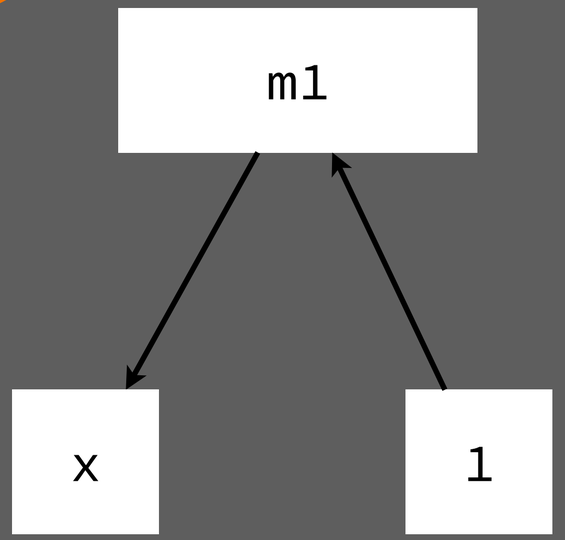
Hardware



Dahlia

```
let m1: float[10];

let x = m1[0];
---
m1[1] := 1;
```

Hardware

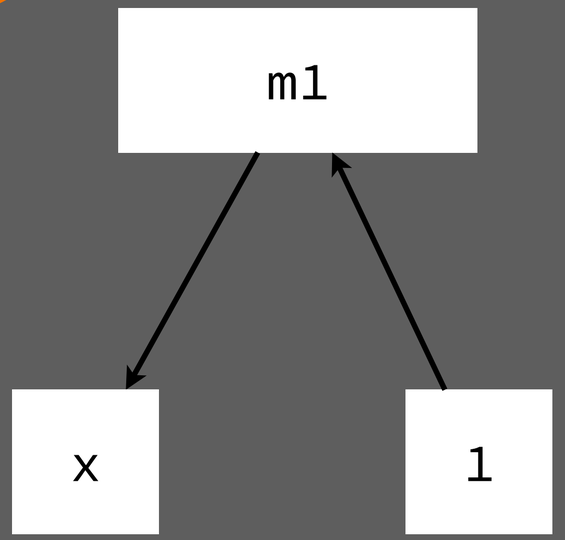


Dahlia

```
let m1: float[10];

let x = m1[0];
---
m1[1] := 1;
```

Ordered Composition

Hardware



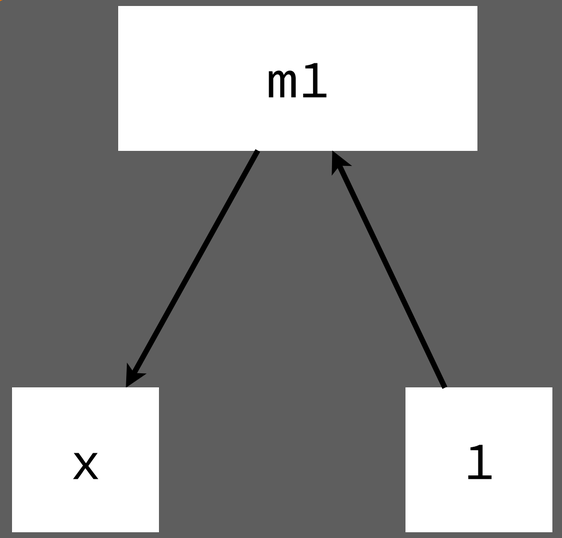
Dahlia

```
let m1: float[10];

let x = m1[0];
---
m1[1] := 1;
```

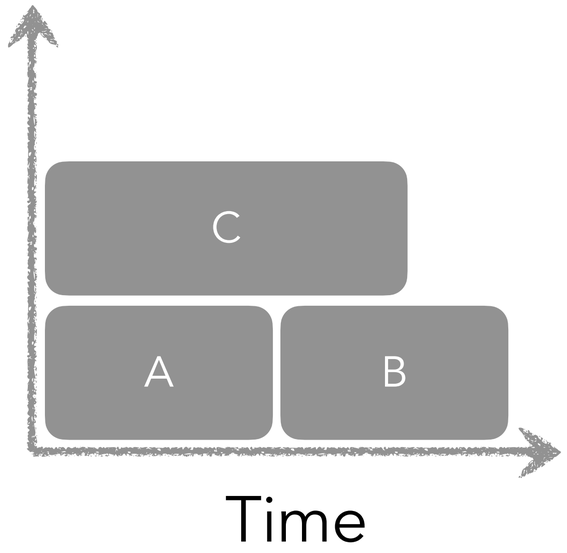
Ordered Composition

- Run sequentially.
- Renew affine resources.

Hardware

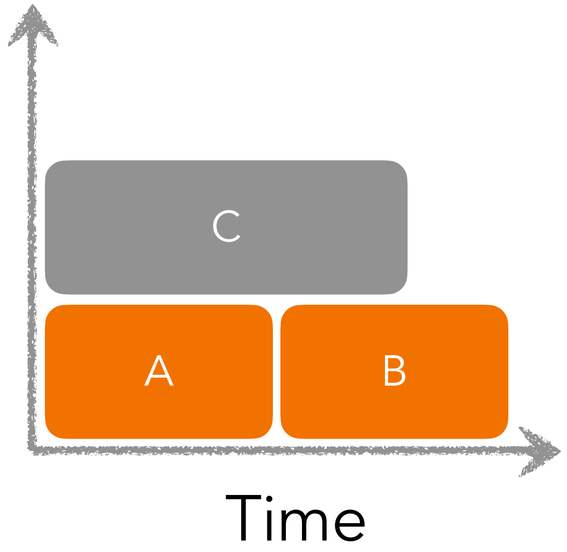


```
{ A --- B };
C
```



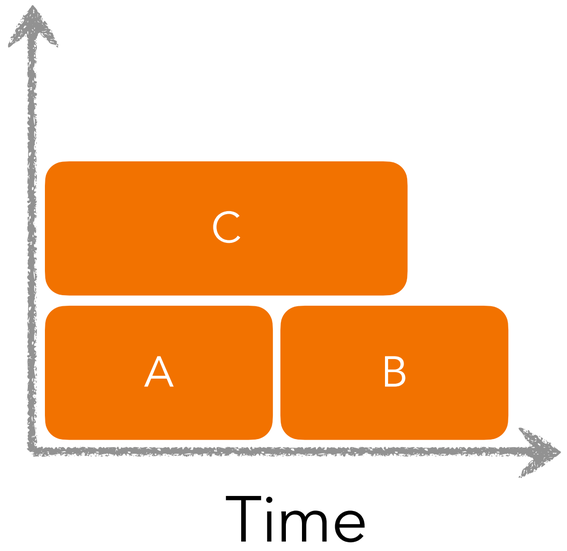
Temporally exclusive use of resources

```
{ A --- B };
C
```



Temporally exclusive use of resources

```
{ A --- B };
C
```

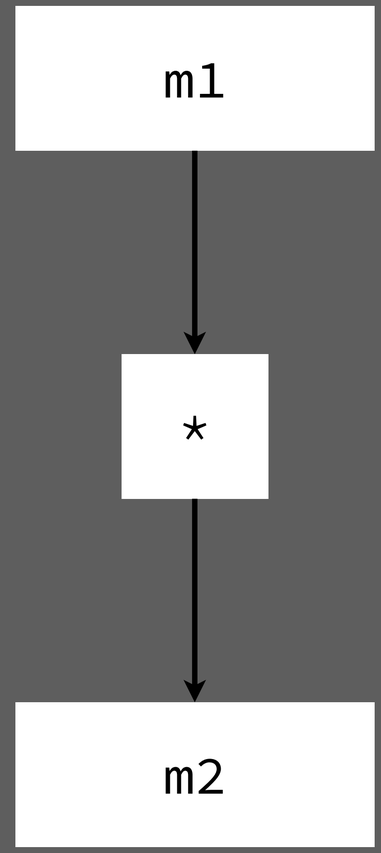


Temporally exclusive use of resources

Dahlia

```
let m1: float[12];
let m2: float[12];

for (let i = 0 .. 12) {
  m2[i] = m1[i] * 2;
}
```

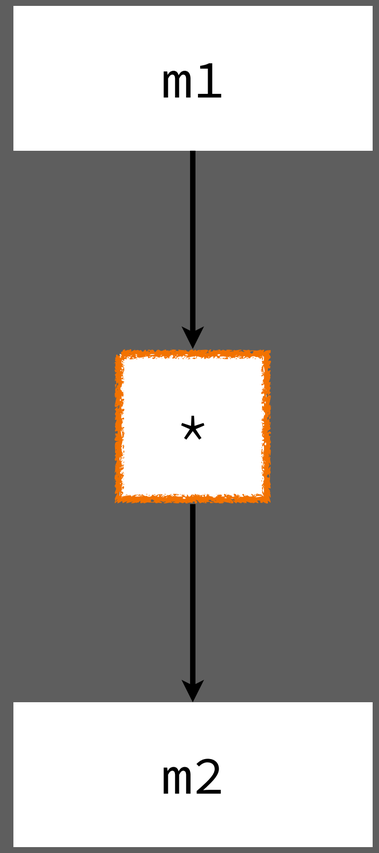
Hardware



Dahlia

```
let m1: float[12];
let m2: float[12];

for (let i = 0 .. 12) unroll 3 {
  m2[i] = m1[i] * 2;
}
```

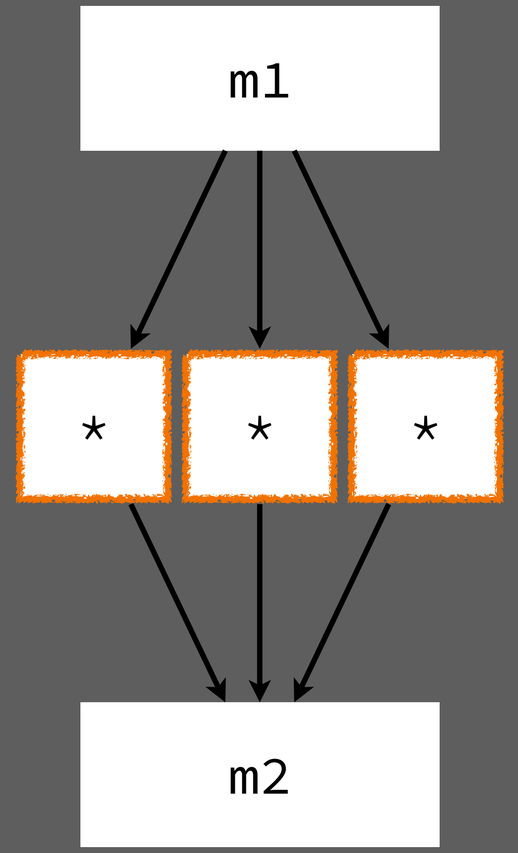
Hardware



Dahlia

```
let m1: float[12];
let m2: float[12];

for (let i = 0 .. 12) unroll 3 {
  m2[i] = m1[i] * 2;
}
```

Hardware

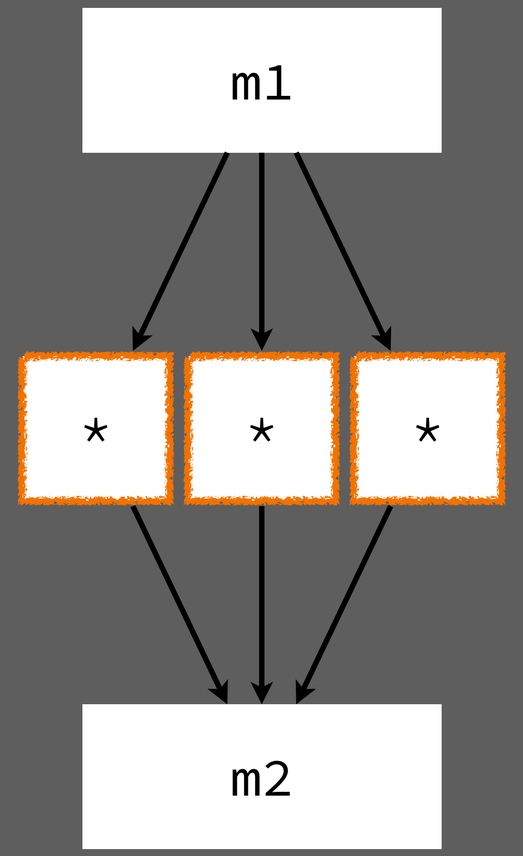


Dahlia

```
let m1: float[12];
let m2: float[12];

for (let i = 0 .. 12) unroll 3 {
  m2[i] = m1[i] * 2;
}
```

Error: Affine resource ‘m1’
already used in this context.

Hardware

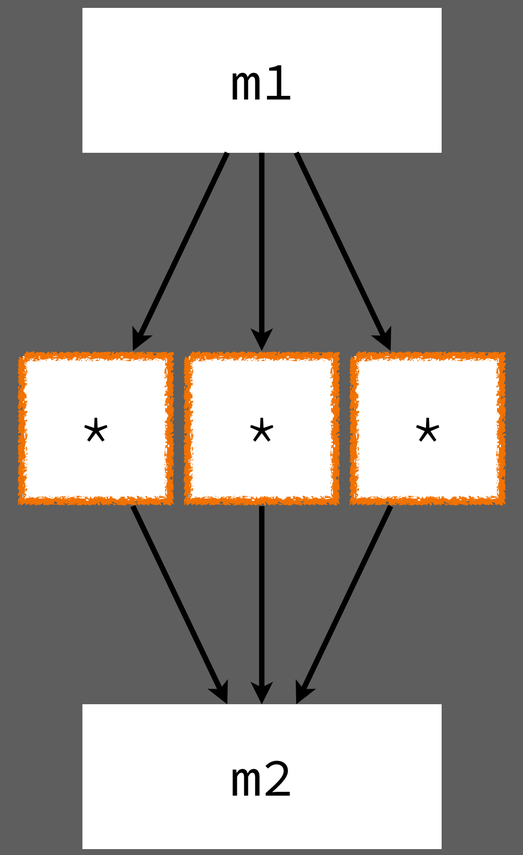


Dahlia

```
let m1: float[12];
let m2: float[12];

for (let i = 0 .. 4) {
  m2[3*i+0] = m1[3*i+0] * 2;
  m2[3*i+1] = m1[3*i+1] * 2;
  m2[3*i+2] = m1[3*i+2] * 2;
}
```

Error: Affine resource ‘m1’
already used in this context.

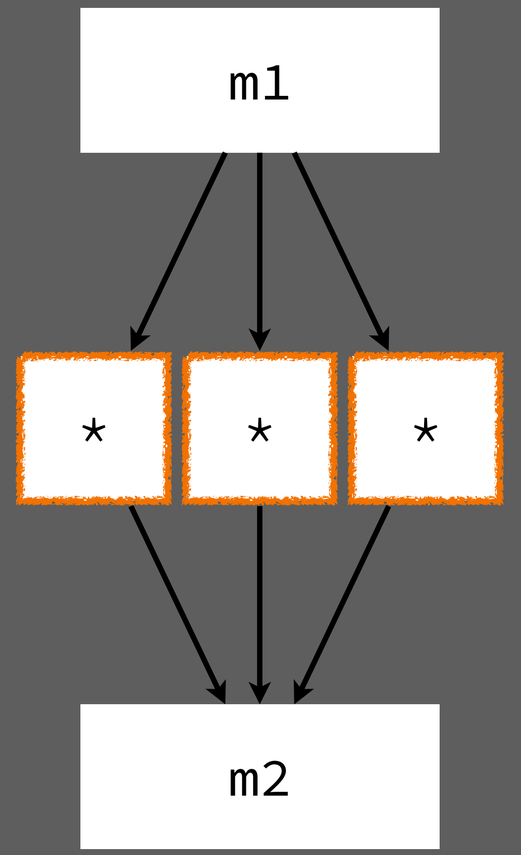
Hardware



Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 12) unroll 3 {
  m2[i] = m1[i] * 2;
}
```

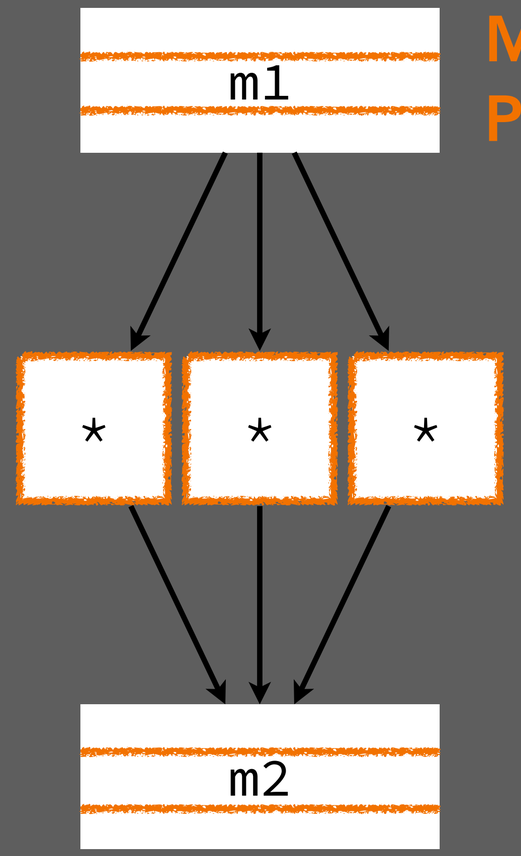
Hardware



Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 12) unroll 3 {
  m2[i] = m1[i] * 2;
}
```

Hardware



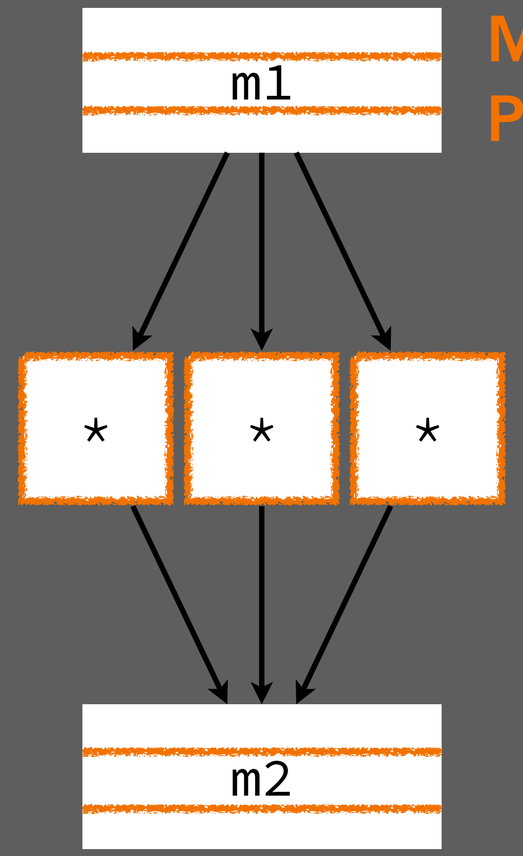
Memory Partitions

Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 12) unroll 3 {
  m2[i] = m1[i] * 2;
}
```

OK: Dahlia guarantees that parallel accesses use disjoint partitions.

Hardware

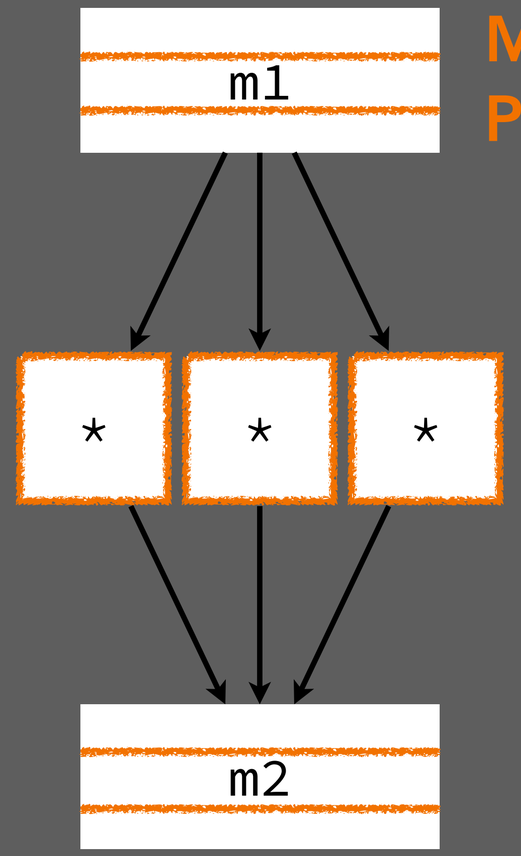


Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {
  m2[3*i+0] = m1[3*i+0] * 2;
  m2[3*i+1] = m1[3*i+1] * 2;
  m2[3*i+2] = m1[3*i+2] * 2;
}
```

OK: Dahlia guarantees that parallel accesses use disjoint partitions.

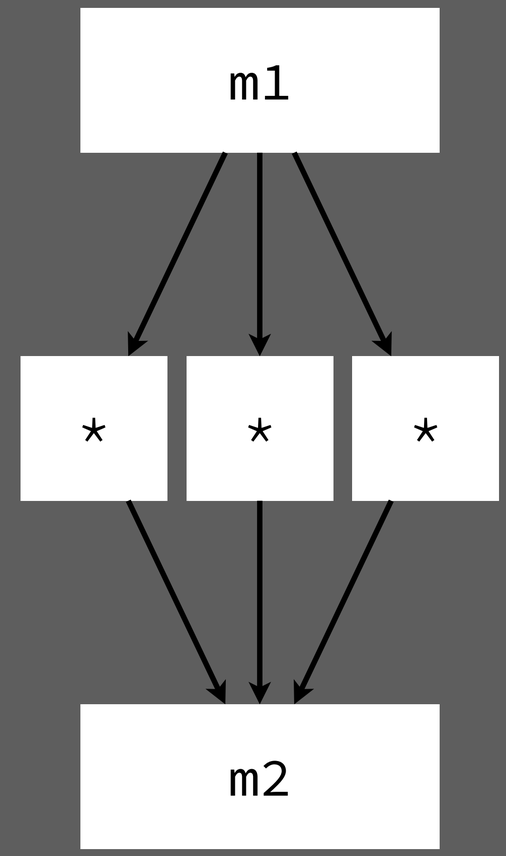
Hardware



Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {
  for (let j = 0 .. 3) {
    m2[i] = m1[3*i+j] * 2;
  }
}
```

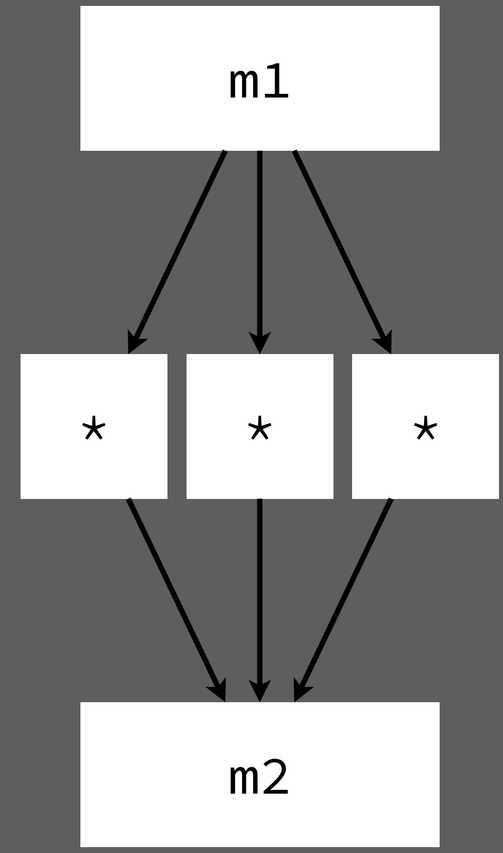
Hardware



Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {
  for (let j = 0 .. 3) unroll 3 {
    m2[i] = m1[3*i+j] * 2;
  }
}
```

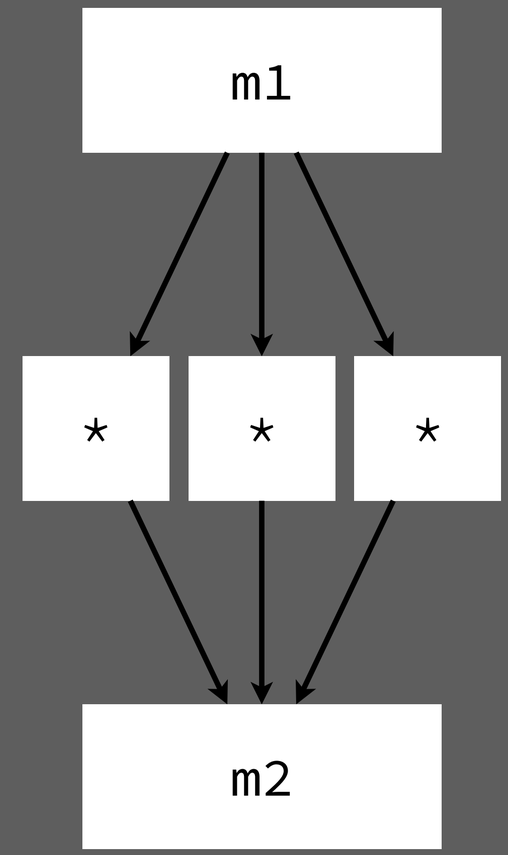
Hardware



Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {
  for (let j = 0 .. 3) unroll 3 {
    m2[i] = m1[f(i, j)] * 2;
  }
}
```

Hardware

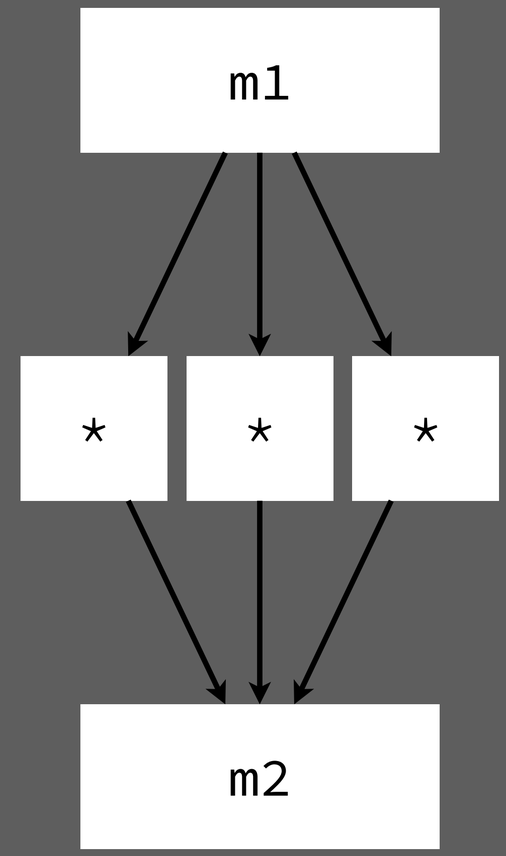


Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {
  for (let j = 0 .. 3) unroll 3 {
    m2[i] = m1[f(i, j)] * 2;
  }
}
```

Parallelizing access patterns requires
unpredictable analyses

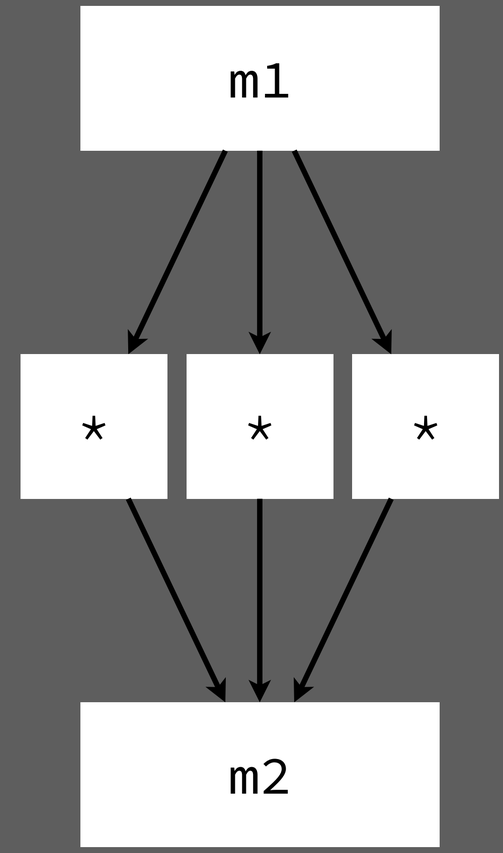
Hardware



Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {
  for (let j = 0 .. 3) unroll 3 {
    m2[i] = m1[3*i+j] * 2;
  }
}
```

Hardware

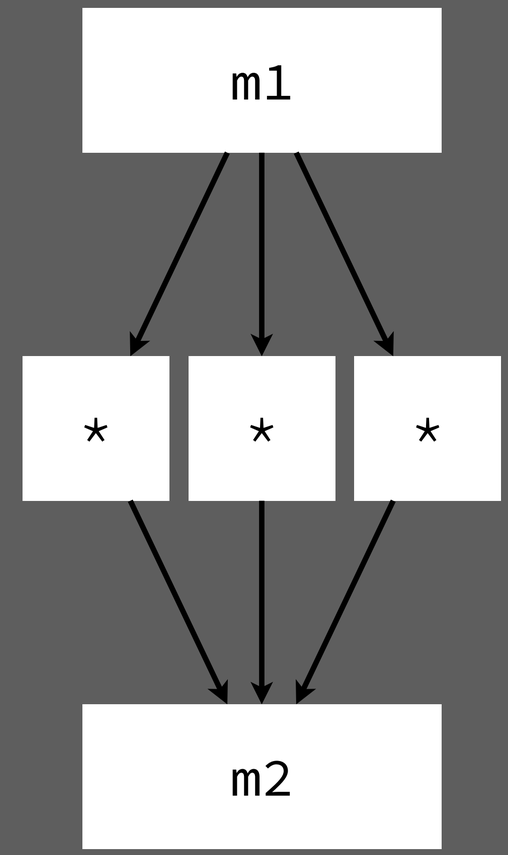


Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {
  for (let j = 0 .. 3) unroll 3 {
    m2[i] = m1[3*i+j] * 2;
  }
}
```

Error: Cannot parallelize dynamic access pattern.

Hardware



Memory **Views**

Memory **Views**

```
let m1: float[12 bank 3];
```

Memory **Views**

```
let m1: float[12 bank 3];

view v1 = suffix m1[by 2*i];
```

Memory **Views**

```
let m1: float[12 bank 3];

view v1 = suffix m1[by 2*i];
```

- Hardware cost of indexing logic

Memory **Views**

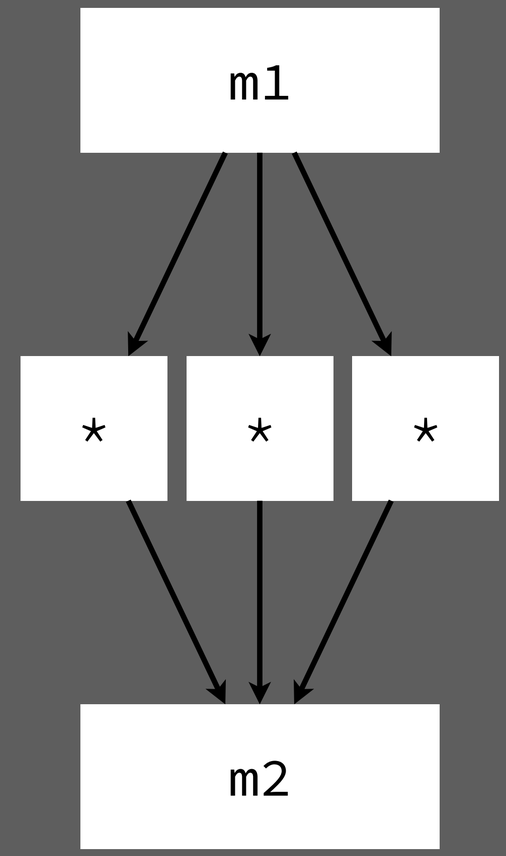
```
let m1: float[12 bank 3];

view v1 = suffix m1[by 2*i];
```

- Hardware cost of indexing logic
- Proof that accesses can be parallelized

Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {

  view v1 = suffix m1[by 3*i];

  for (let j = 0 .. 3) unroll 3 {
    m2[i] = v1[j] * 2; // m1[3*i+j]
  }
}
```

Hardware

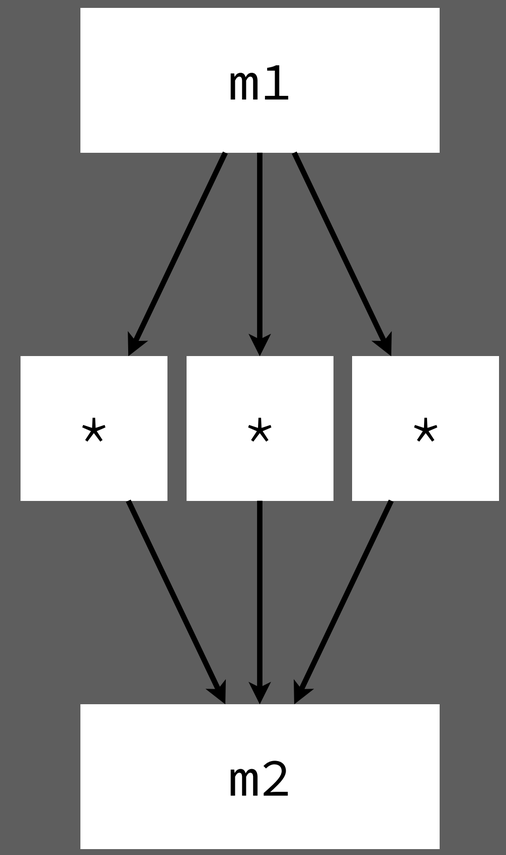


Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {

  view v1 = suffix m1[by 3*i];

  for (let j = 0 .. 3) unroll 3 {
    m2[i] = v1[j] * 2; // m1[3*i+j]
  }
}
```

Suffix View

Hardware



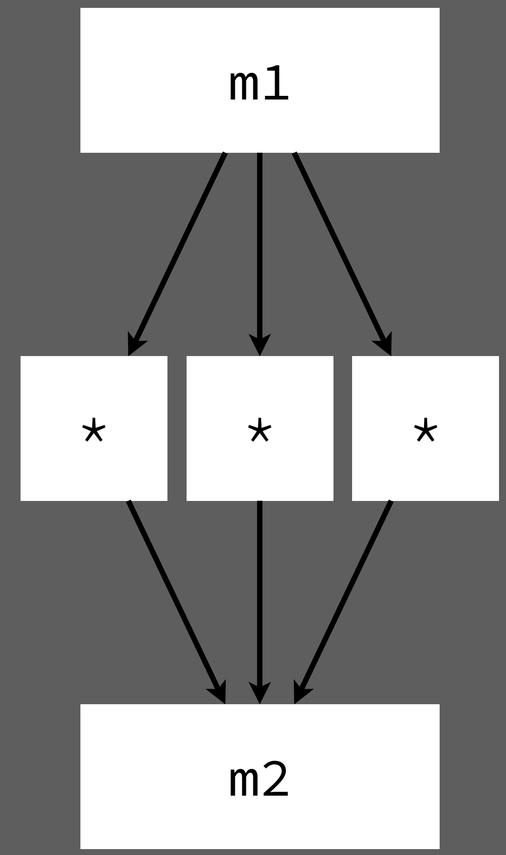
Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {

  view v1 = suffix m1[by 3*i];

  for (let j = 0 .. 3) unroll 3 {
    m2[i] = v1[j] * 2; // m1[3*i+j]
  }
}
```

Suffix View

- Offset by banking factor

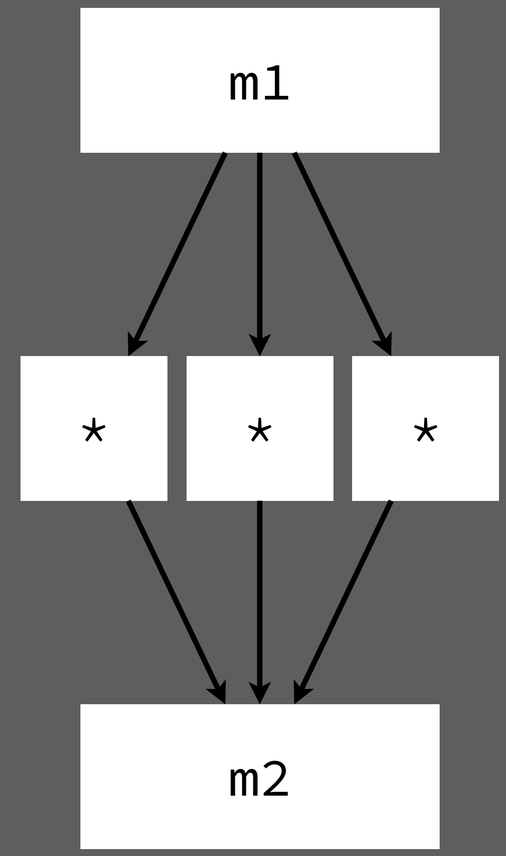
Hardware



Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {

  view v1 = shift m1[by f(i)];

  for (let j = 0 .. 3) unroll 3 {
    m2[i] = v1[j] * 2; // m1[f(i)+j]
  }
}
```

Hardware

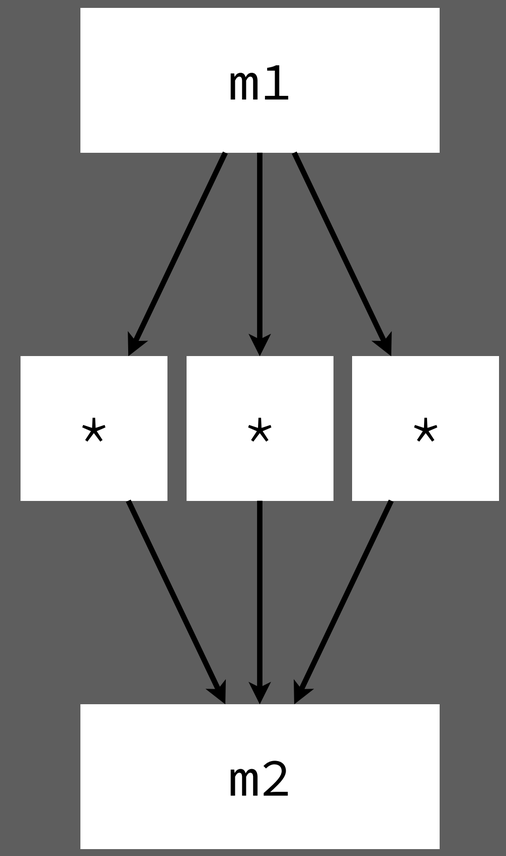


Dahlia

```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {

  view v1 = shift m1[by f(i)];

  for (let j = 0 .. 3) unroll 3 {
    m2[i] = v1[j] * 2; // m1[f(i)+j]
  }
}
```

Shift View

Hardware



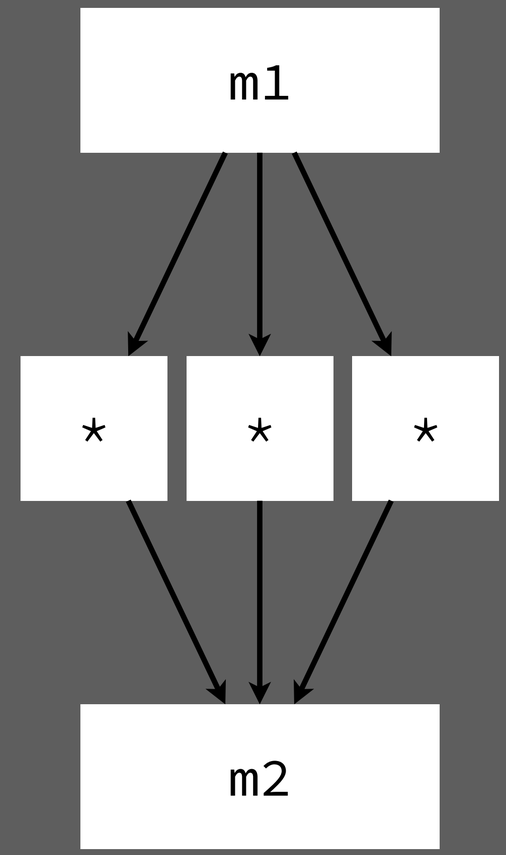
Dahlia

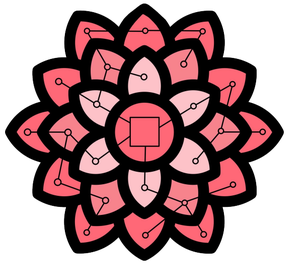
```
let m1: float[12 bank 3];
let m2: float[12 bank 3];

for (let i = 0 .. 4) {

  view v1 = shift m1[by f(i)];

  for (let j = 0 .. 3) unroll 3 {
    m2[i] = v1[j] * 2; // m1[f(i)+j]
  }
}
```

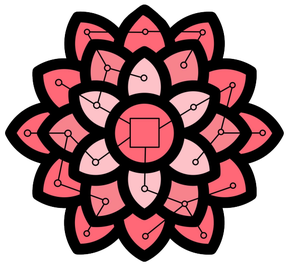
Shift View

- Arbitrary offset

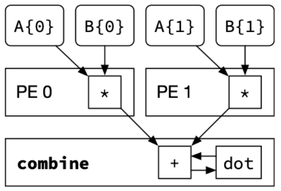
Hardware

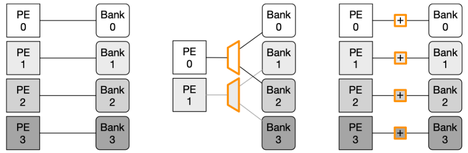


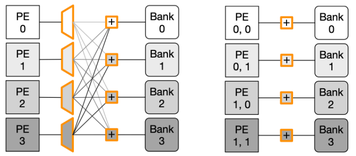


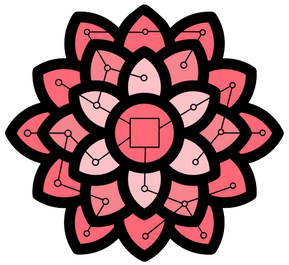


Language Features

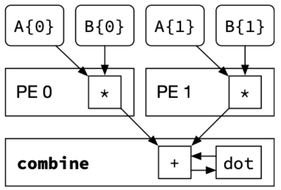


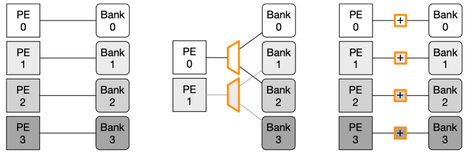


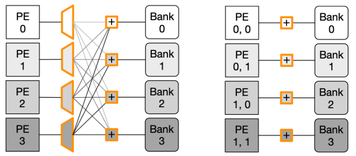




Language Features







Formalism

$x \in \text{variables} \quad a \in \text{memories} \quad n \in \text{numbers}$

$b ::= \textbf{true} \mid \textbf{false} \qquad v ::= n \mid b$

$e ::= v \mid \textbf{bop}\ e_1\ e_2 \mid x \mid a[e]$

$c ::= e \mid \textbf{let}\ x = e \mid c_1 \multimap c_2 \mid c_1\ ;\ c_2 \mid \textbf{if}\ x\ c_1\ c_2 \mid$
$\textbf{while}\ x\ c \mid x := e \mid a[e_1] := e_2 \mid \textbf{skip}$

$\tau ::= \textbf{bit}\langle n \rangle \mid \textbf{float} \mid \textbf{bool} \mid \textbf{mem}\ \tau[n_1]$

$$\boxed{\sigma_1, \rho_1, c \to \sigma', \rho', c'}$$

$$\frac{\sigma, \rho, e_1 \to \sigma', \rho', e_1'}{\sigma, \rho, a[e_1] := e_2 \to \sigma', \rho', a[e_1'] := e_2}$$

$$\frac{\sigma, \rho, e \to \sigma', \rho', e'}{\sigma, \rho, a[n] := e \to \sigma', \rho', a[n] := e'}$$

$$\frac{a \notin \rho}{\sigma, \rho, a[n] := v \to \sigma[a[n] \mapsto v], \rho \cup \{a\}, \textbf{skip}}$$

$$\frac{\sigma, \rho, e \to \sigma', \rho', e'}{\sigma, \rho, \textbf{let}\ x = e \to \sigma', \rho', \textbf{let}\ x = e'}$$

$$\frac{}{\sigma, \rho, \textbf{let}\ x = v \to \sigma[x \mapsto v], \rho, \textbf{skip}}$$

$$\frac{\sigma, \rho, c_1 \to \sigma', \rho', c_1'}{\sigma, \rho, c_1\ ;\ c_2 \to \sigma', \rho', c_1'\ ;\ c_2} \qquad \frac{}{\sigma, \rho, \textbf{skip}\ ;\ c_2 \to \sigma, \rho, c_2}$$

$$\frac{}{\sigma, \rho, c_1 \multimap c_2 \to \sigma, \rho, c_1 \overset{\rho}{\sim} c_2}$$

$$\frac{\sigma, \rho, c_1 \to \sigma', \rho', c_1'}{\sigma, \rho, c_1 \overset{\rho''}{\sim} c_2 \to \sigma', \rho', c_1' \overset{\rho''}{\sim} c_2}$$

$$\frac{\sigma, \rho'', c_2 \to \sigma', \rho''', c_2'}{\sigma, \rho, \textbf{skip} \overset{\rho''}{\sim} c_2 \to \sigma', \rho, \textbf{skip} \overset{\rho'''}{\sim} c_2'}$$

$$\frac{}{\sigma, \rho, \textbf{skip} \overset{\rho''}{\sim} \textbf{skip} \to \sigma, \rho \cup \rho'', \textbf{skip}}$$

$$\boxed{\sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, v}$$

$$\frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, v_1 \qquad \sigma_2, \rho_2, e_2 \Downarrow \sigma_3, \rho_3, v_2 \qquad v_3 = v_1\ \textbf{bop}\ v_2}{\sigma_1, \rho_1, \textbf{bop}\ e_1\ e_2 \Downarrow \sigma_3, \rho_3, v_3}$$

$$\frac{\sigma(x) = v}{\sigma, \rho, x \Downarrow \sigma, \rho, v}$$

$$\frac{a \notin \rho_1 \qquad \sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, n \qquad \sigma_2(a)(n) = v}{\sigma_1, \rho_1, a[e] \Downarrow \sigma_2, \rho_2 \cup \{a\}, v}$$

$$\boxed{\sigma_1, \rho_1, c \Downarrow \sigma_2, \rho_2}$$

$$\frac{\sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, v}{\sigma_1, \rho_1, \textbf{let}\ x = e \Downarrow \sigma_2[x \mapsto v], \rho_2}$$

$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_1, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1 \multimap c_2 \Downarrow \sigma_3, \rho_2 \cup \rho_3}$$

$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1 \overset{\rho}{\sim} c_2 \Downarrow \sigma_3, \rho_2 \cup \rho_3}$$

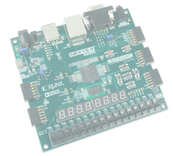
$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_2, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1\ ;\ c_2 \Downarrow \sigma_3, \rho_3}$$

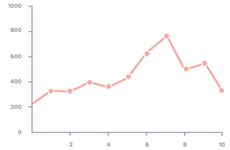
$$\frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, \textbf{true} \qquad \sigma_2, \rho_2, c_1 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, \textbf{if}\ x\ c_1\ c_2 \Downarrow \sigma_3, \rho_3}$$

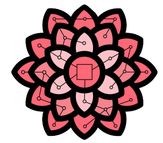
$$\frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, \textbf{false} \qquad \sigma_2, \rho_2, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, \textbf{if}\ x\ c_1\ c_2 \Downarrow \sigma_3, \rho_3}$$

$$\frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, \textbf{true} \quad \sigma_2, \rho_2, c \multimap \textbf{while}\ x\ c \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, \textbf{while}\ x\ c \Downarrow \sigma_3, \rho_3} \qquad \frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, \textbf{false}}{\sigma_1, \rho_1, \textbf{while}\ x\ c \Downarrow \sigma_2, \rho_2}$$

Ada's Journey







Type Systems

Well-typed programs
cannot **go wrong**

Type Systems

Well-typed programs
make **predictable trade-offs**

Design Space Exploration

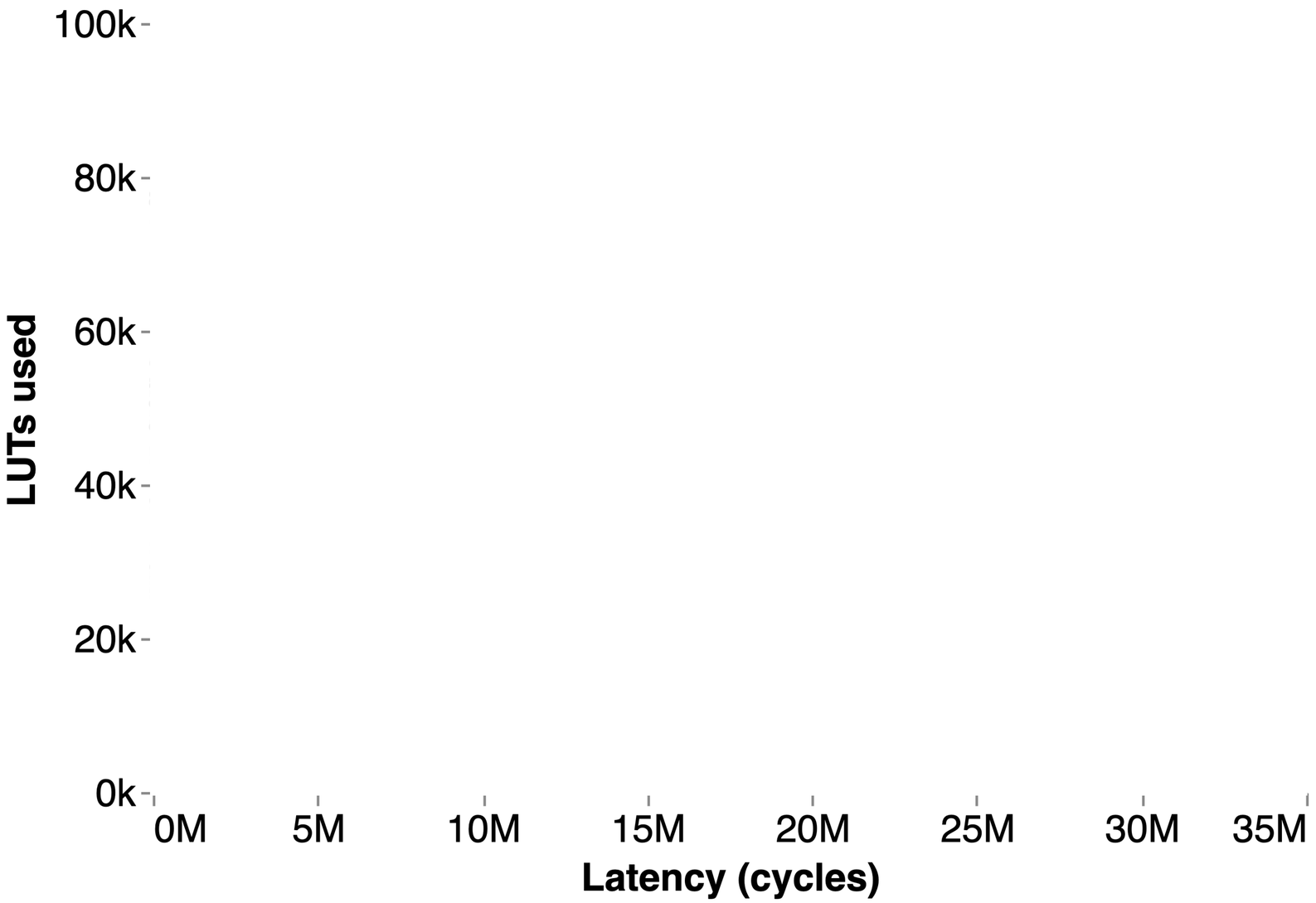
Design Space Exploration

Blocked matrix-matrix multiply

32,000 designs

2,600 hours of *estimation*

Design Space Exploration

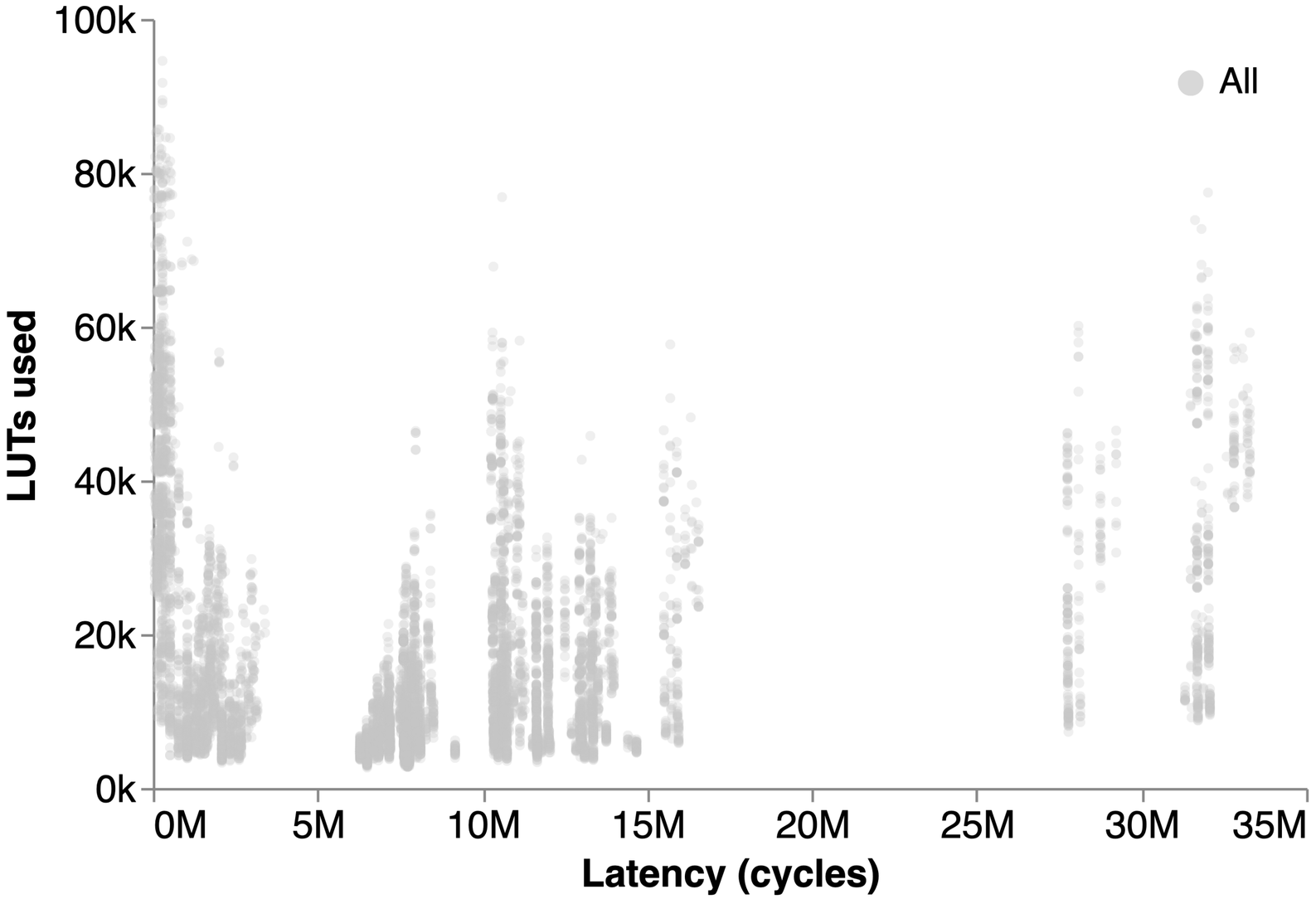


Blocked matrix-matrix multiply

32,000 designs

2,600 hours of *estimation*

Design Space Exploration

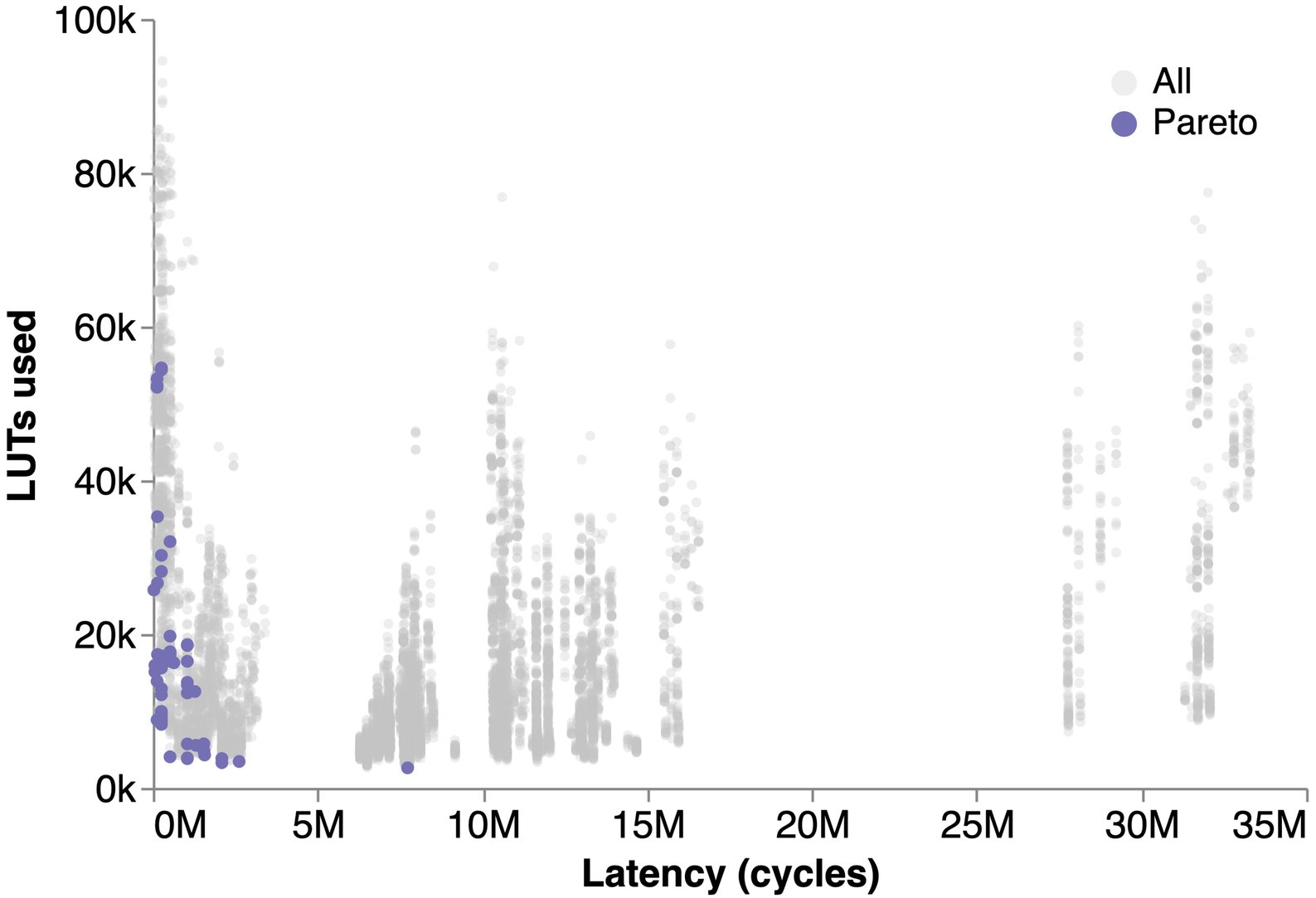


Blocked matrix-matrix multiply

32,000 designs

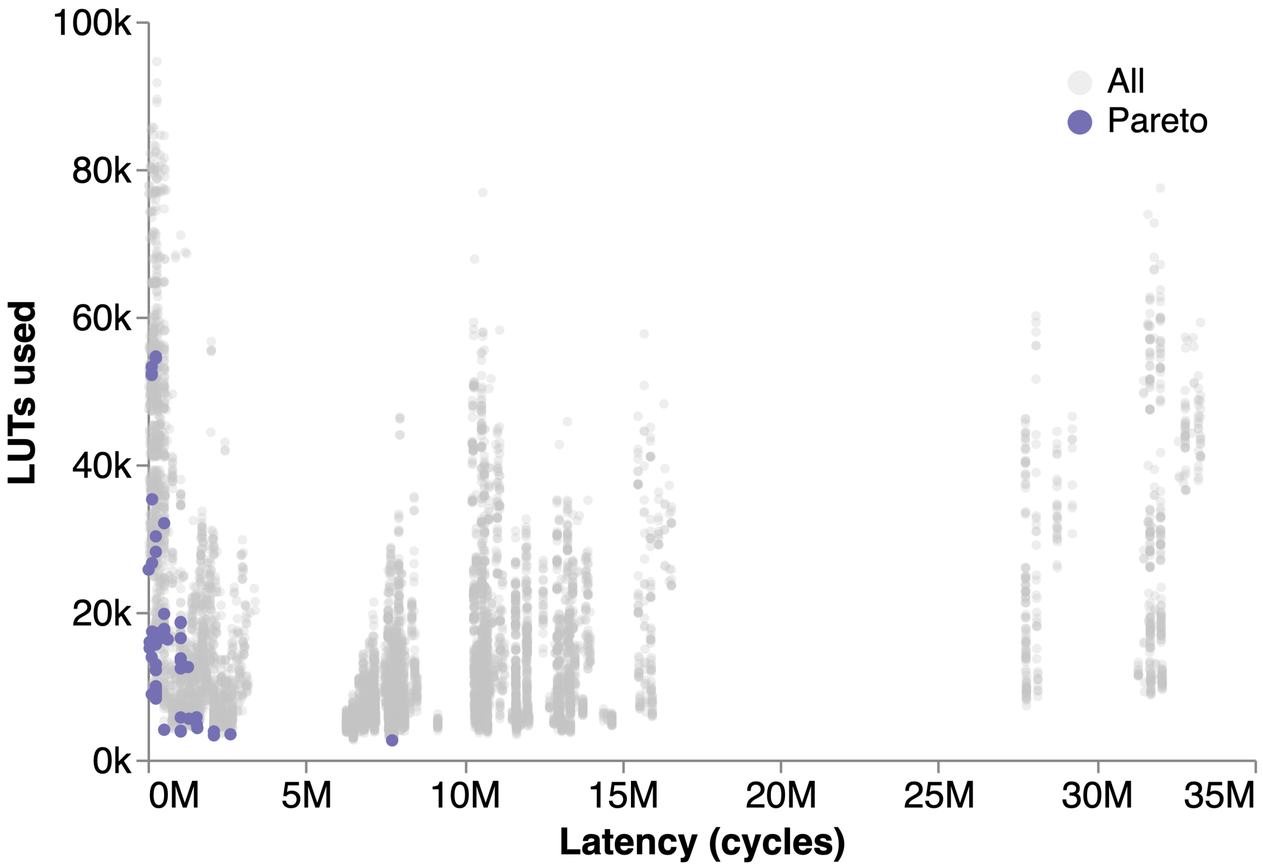
2,600 hours of *estimation*

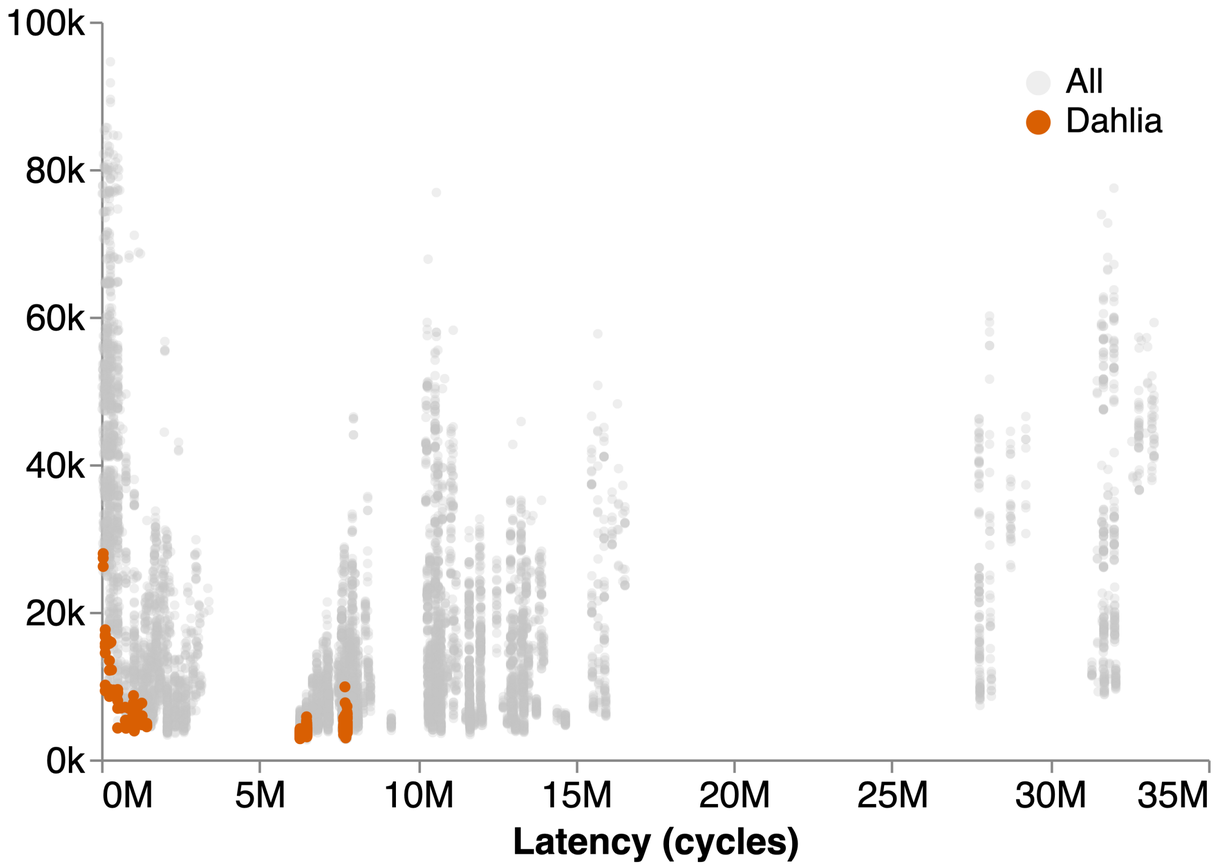
Design Space Exploration



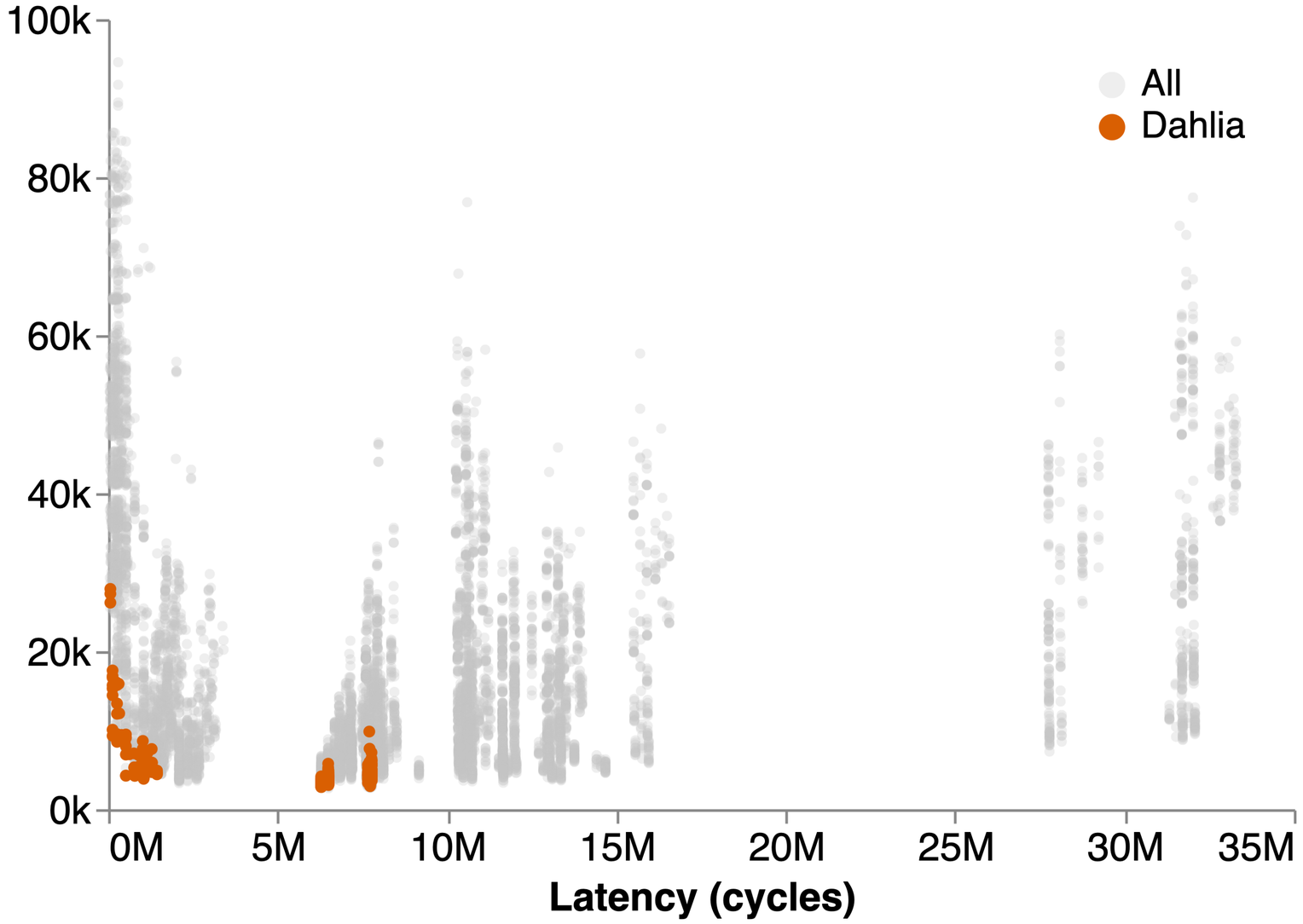
Pareto optimal w.r.t FPGA resources and runtime

Design Space Exploration



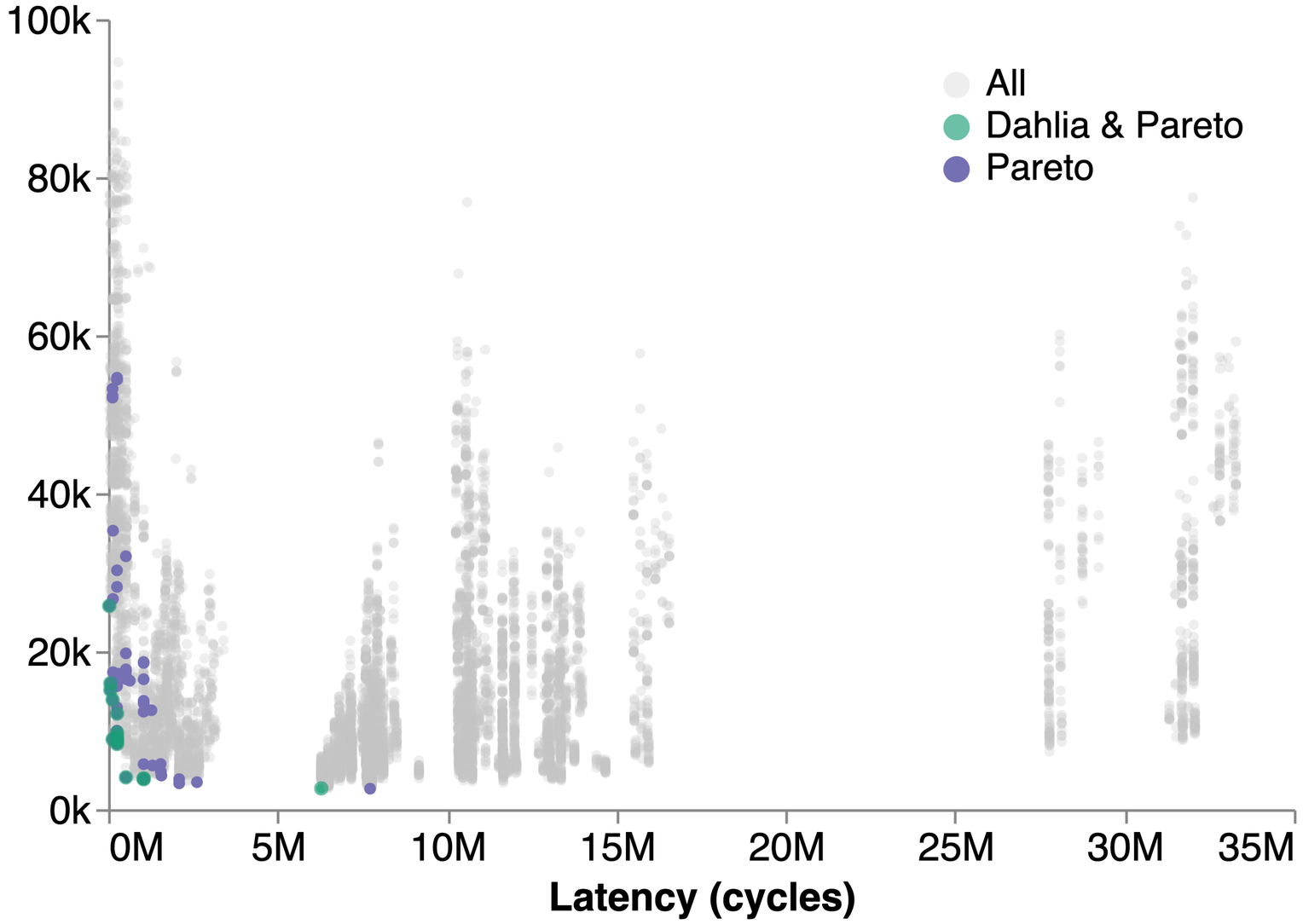


Design Space Exploration



354 out of **32,000** configurations accepted

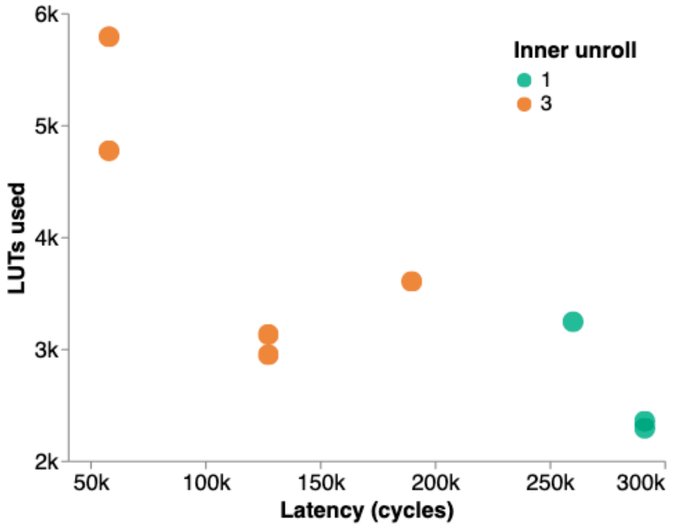
Design Space Exploration



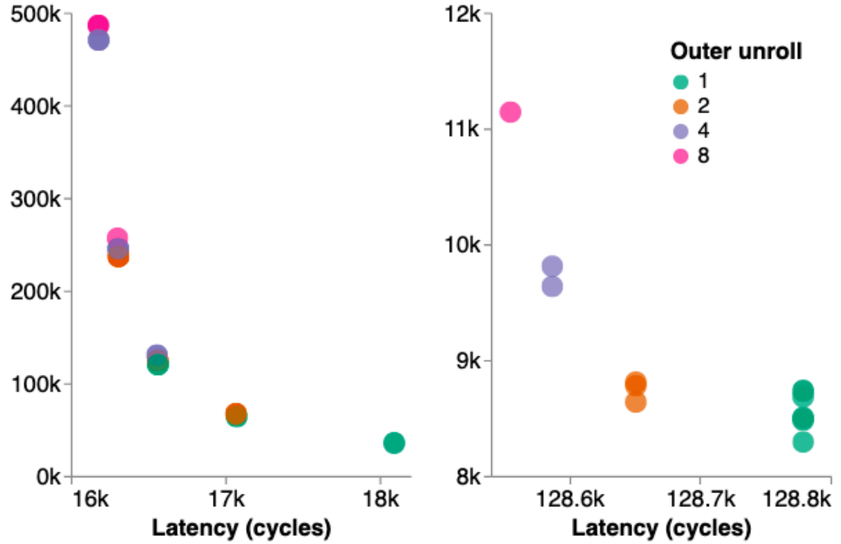
354 out of **32,000** configurations accepted

15 out of **56 Pareto** configurations accepted

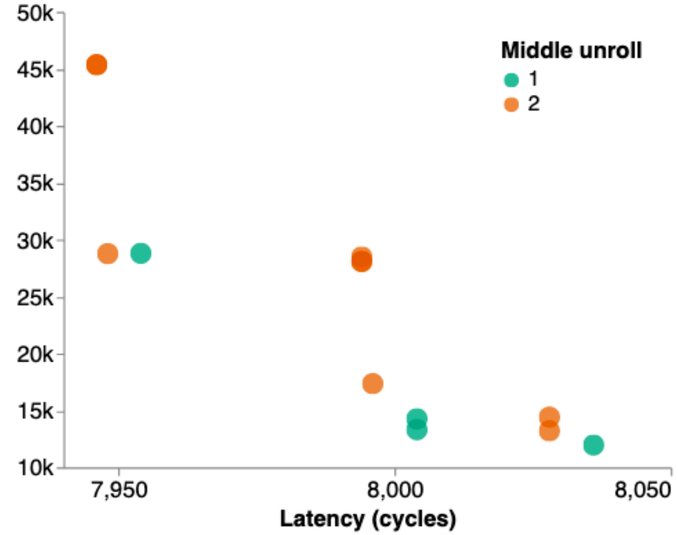
Qualitative Studies



(a) stencil2d with inner unroll.



(b) md-knn with outer unroll.



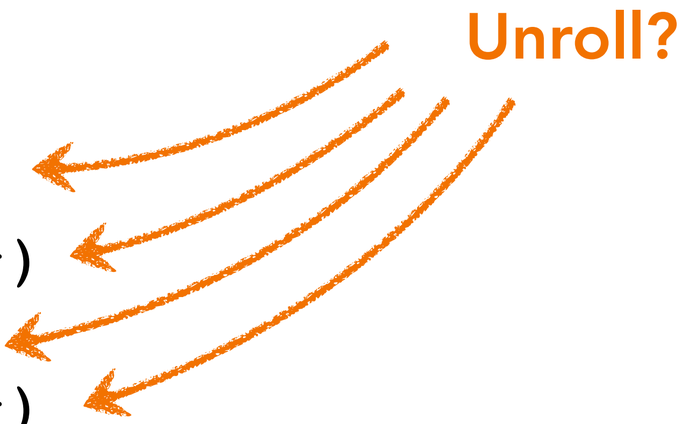
(c) md-grid with middle unroll.

Qualitative Studies

```
for (r=0; r<row_size; r++)
  for (c=0; c<col_size; c++)
    for (k1=0; k1<3; k1++)
      for (k2=0; k2<3; k2++)
        mul = filter[k1*3 + k2] *
              orig[(r+k1)*col_size + c+k2]
```

2D-Stencil

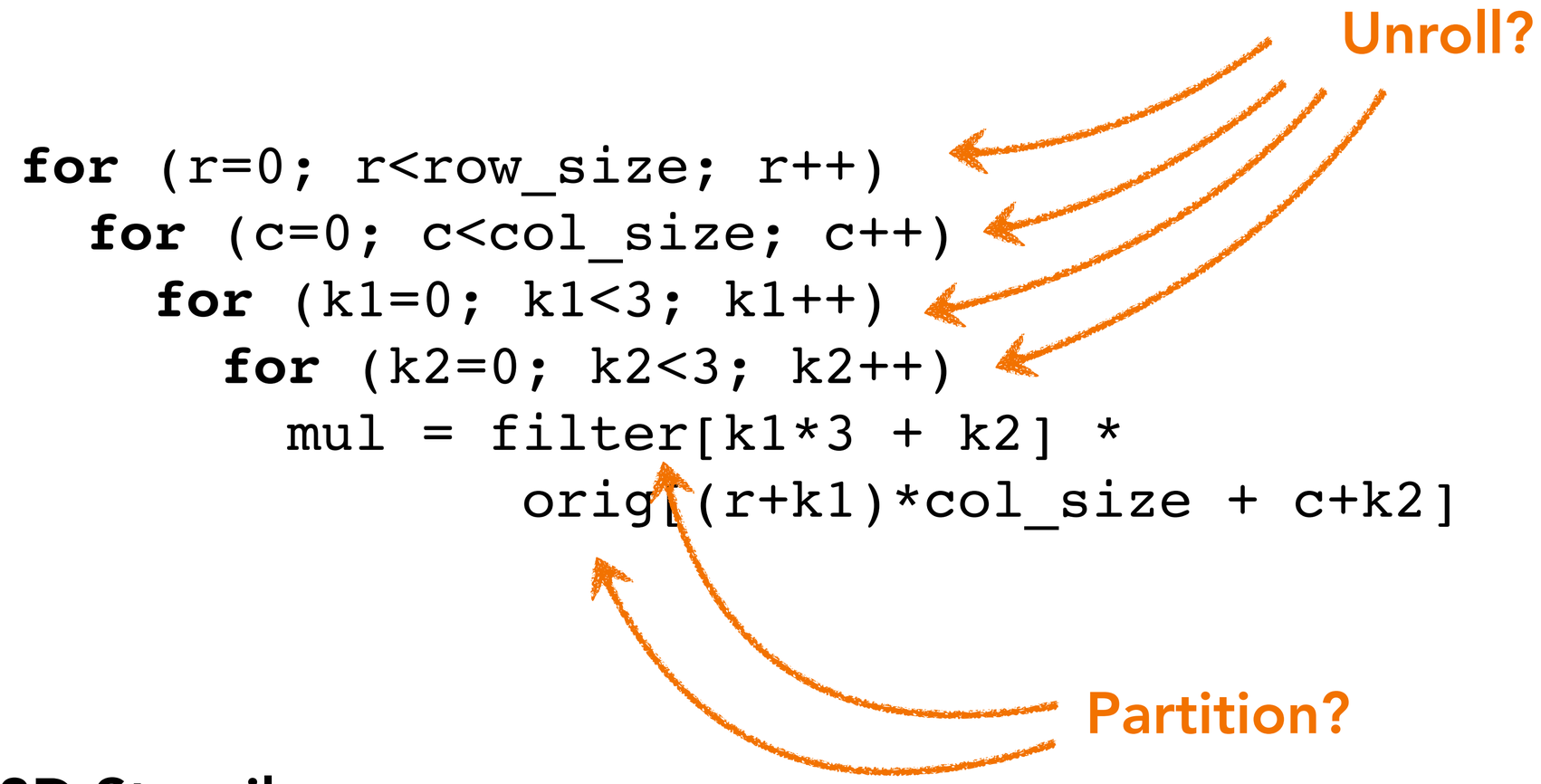
Qualitative Studies




```
for (r=0; r<row_size; r++)
  for (c=0; c<col_size; c++)
    for (k1=0; k1<3; k1++)
      for (k2=0; k2<3; k2++)
        mul = filter[k1*3 + k2] *
              orig[(r+k1)*col_size + c+k2]
```

2D-Stencil

Qualitative Studies



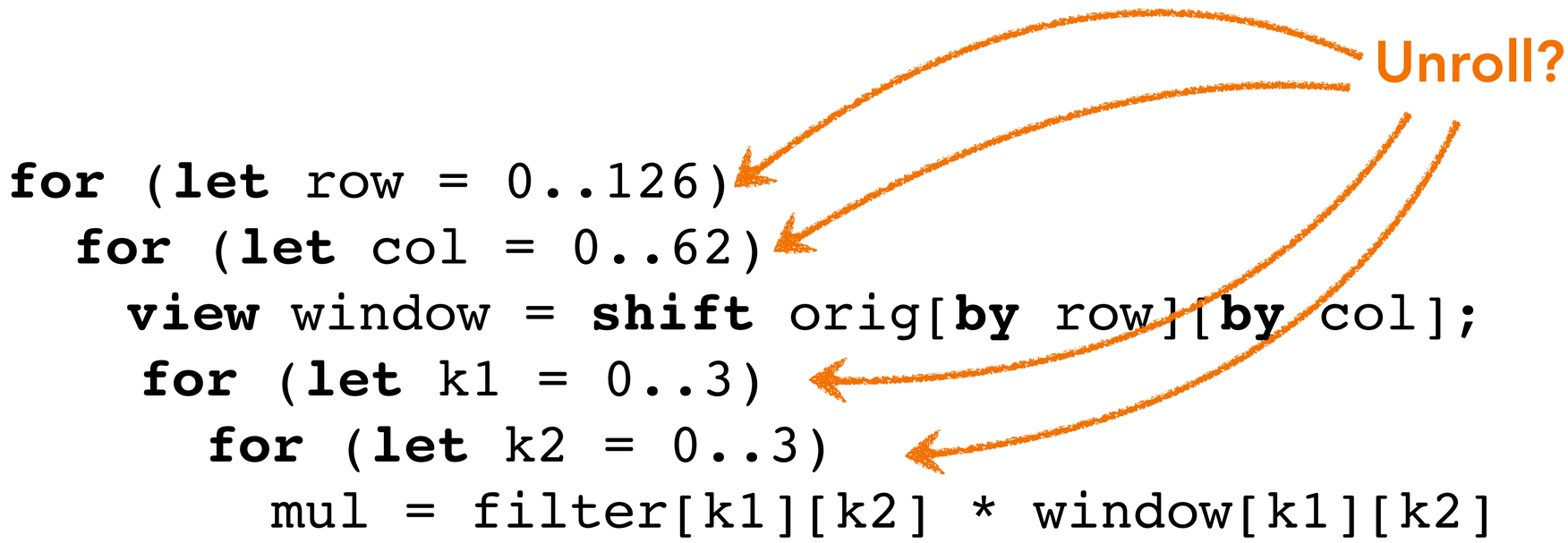
2D-Stencil

Qualitative Studies

```
for (let row = 0..126)
  for (let col = 0..62)
    view window = shift orig[by row][by col];
     for (let k1 = 0..3)
       for (let k2 = 0..3)
         mul = filter[k1][k2] * window[k1][k2]
```

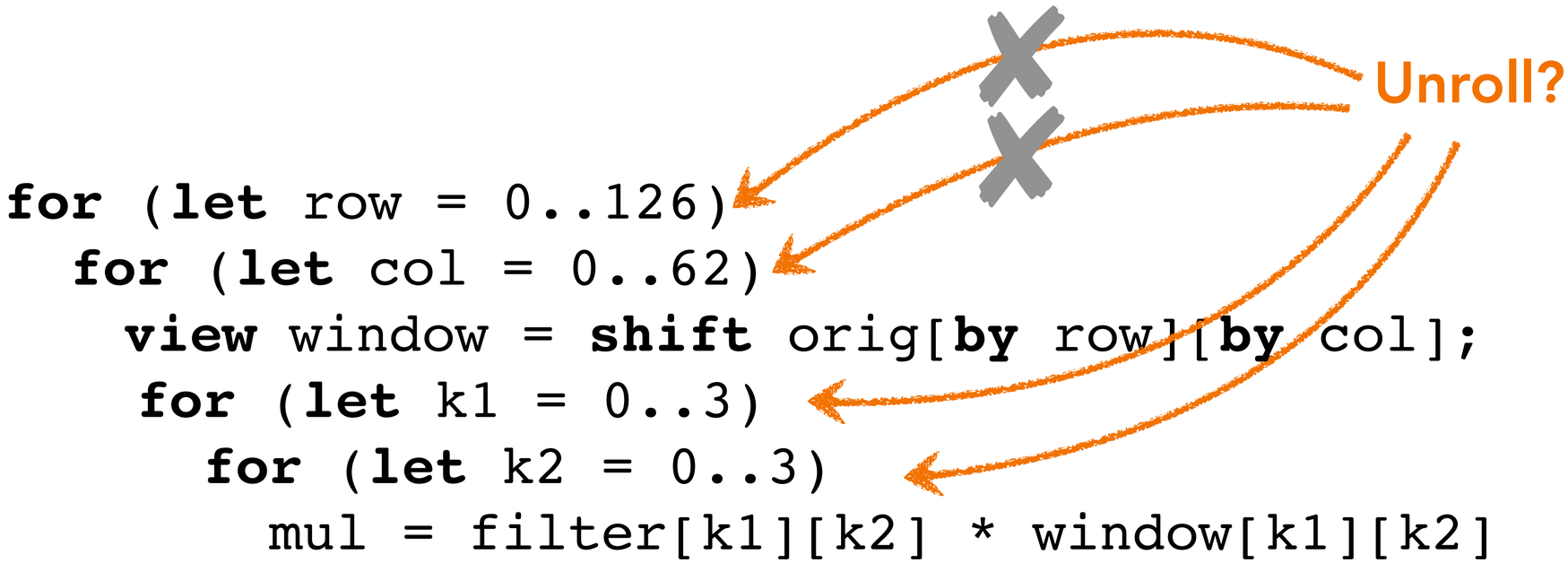
2D-Stencil

Qualitative Studies



2D-Stencil

Qualitative Studies




```
for (let row = 0..126)
  for (let col = 0..62)
    view window = shift orig[by row][by col];
     for (let k1 = 0..3)
       for (let k2 = 0..3)
         mul = filter[k1][k2] * window[k1][k2]
```

2D-Stencil

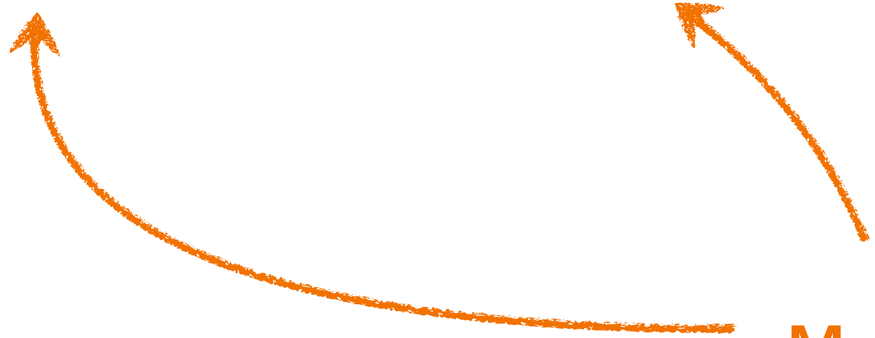
Qualitative Studies

```
for (let row = 0..126)
  for (let col = 0..62)
    view window = shift orig[by row][by col];
     for (let k1 = 0..3) unroll 3
       for (let k2 = 0..3) unroll 3
         mul = filter[k1][k2] * window[k1][k2]
```

2D-Stencil

Qualitative Studies

```
for (let row = 0..126)
  for (let col = 0..62)
    view window = shift orig[by row][by col];
    for (let k1 = 0..3) unroll 3
      for (let k2 = 0..3) unroll 3
        mul = filter[k1][k2] * window[k1][k2]
```



Must partition

2D-Stencil

The Future

The Future

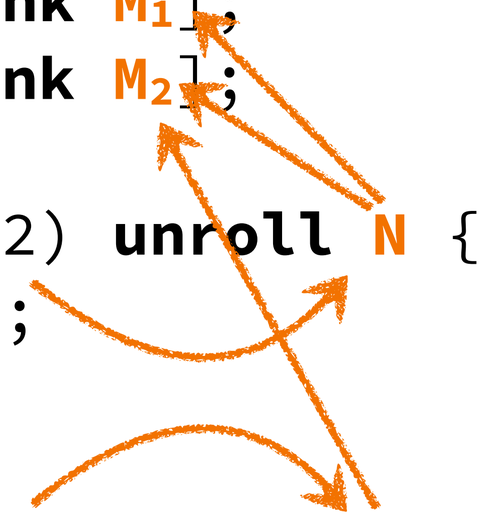
Resource Polymorphism

```
let m1: float[12 bank M1];
let m2: float[12 bank M2];

for (let i = 0 .. 12) unroll N {
  m2[i] = m1[i] * 2;
}

for (let i = 0 .. 12) unroll K {
  sum += m2[i];
}
```

The Future

Resource Polymorphism

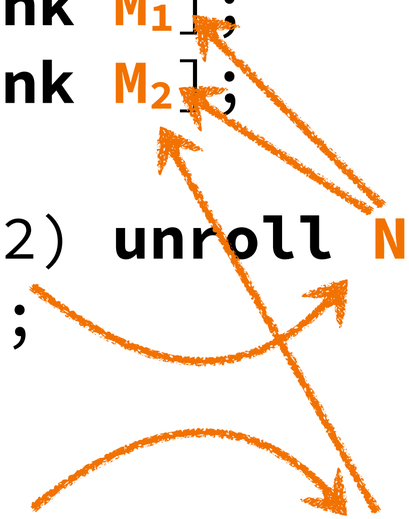
```
let m1: float[12 bank M1];
let m2: float[12 bank M2];

for (let i = 0 .. 12) unroll N {
  m2[i] = m1[i] * 2;
}

for (let i = 0 .. 12) unroll K {
  sum += m2[i];
}
```



The Future

Resource Polymorphism

```
let m1: float[12 bank M1];
let m2: float[12 bank M2];

for (let i = 0 .. 12) unroll N {
  m2[i] = m1[i] * 2;
}

for (let i = 0 .. 12) unroll K {
  sum += m2[i];
}
```



```
M1 = { … }
M2 = { … }
N = { … }
K = { … }
```

The Future

Modularity

The Future

Modularity

```
def dot(m1: float[10], m2: float[10]) {
  let sum = 0;
  for (let i = 0 .. 10) {
    sum += m1[i] * m2[i];
  }
  return sum;
}

dot(A, B);
```

The Future

Modularity

```
def dot(m1: float[10 bank 5], m2: float[10 bank 5]) {
  let sum = 0;
  for (let i = 0 .. 10) unroll 5 {
    sum += m1[i] * m2[i];
  }
  return sum;
}

dot(A, B); // A, B need exactly 5 banks
```

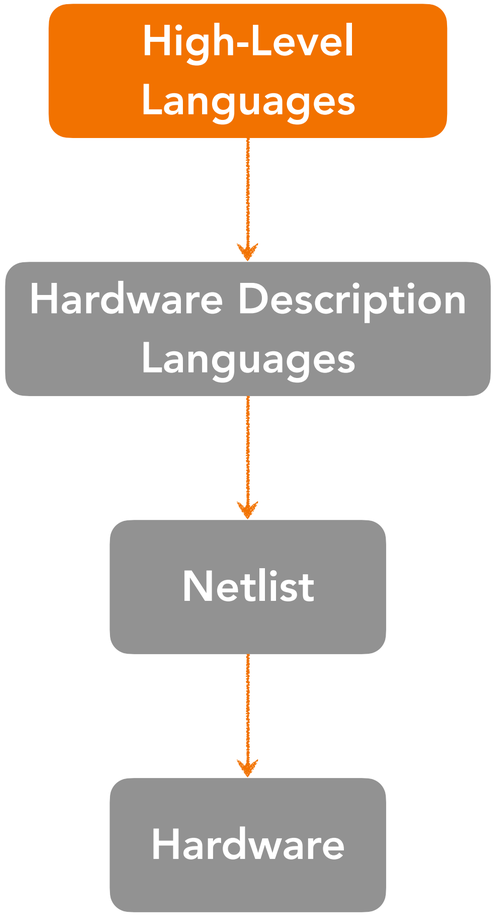
The Future

Modularity

```
def dot(m1: float[10 bank 5], m2: float[10 bank 5]) {
  let sum = 0;
  for (let i = 0 .. 10) unroll 5 {
    sum += m1[i] * m2[i];
  }
  return sum;
}

for (let i = 0..2) unroll 2 {
   dot(A, B); // How many banks? How many copies of dot?
}
```

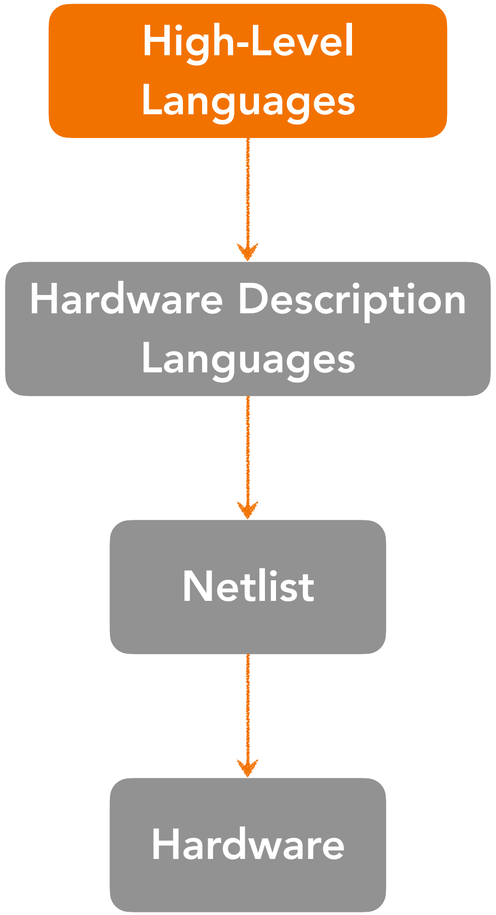
The Future

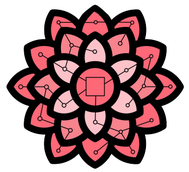
A Predictable Stack



The Future

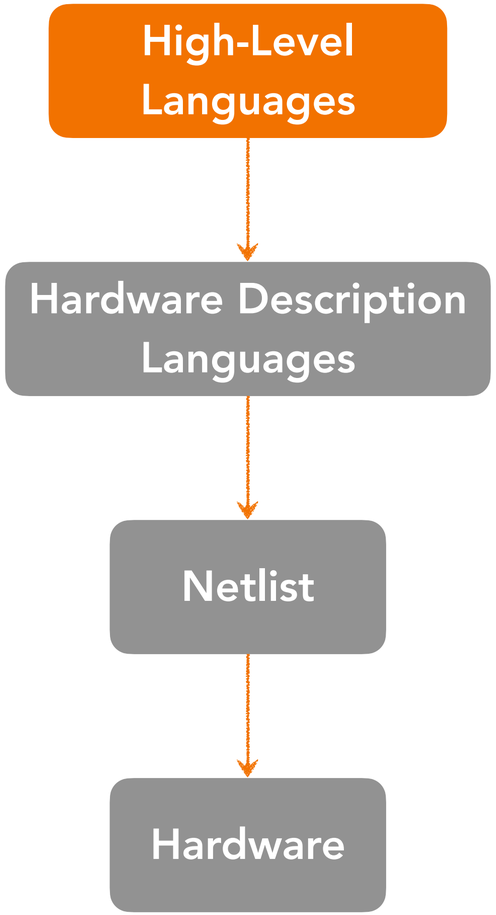
A Predictable Stack

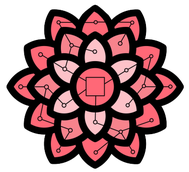




The Future

A Predictable Stack

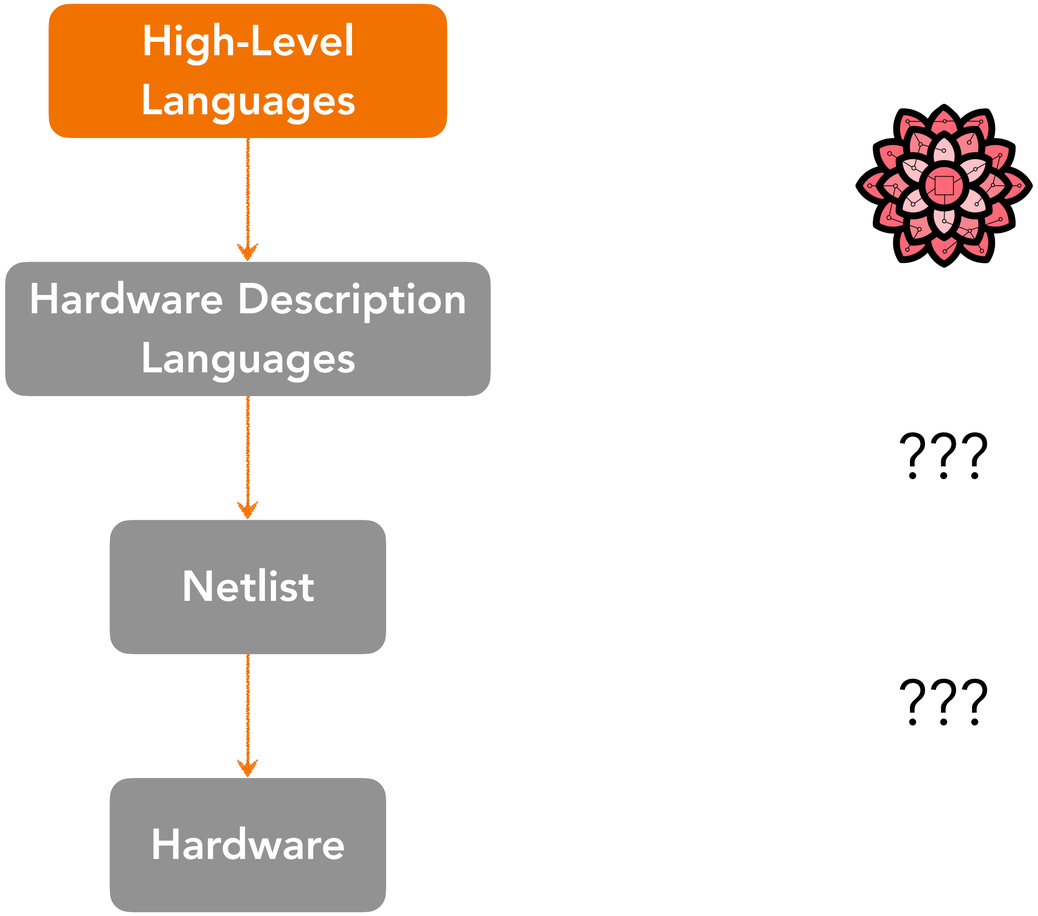




???

The Future

A Predictable Stack



???

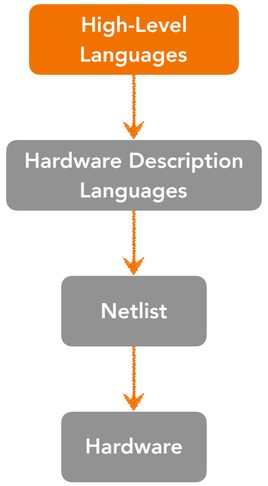
???

The Future

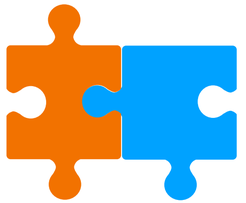
Resource Polymorphism

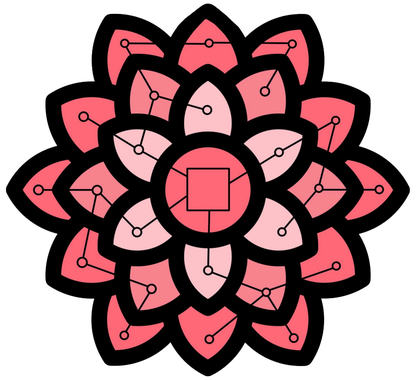
```
let m1: float[12 bank M];
let m2: float[12 bank M];

for (let i = 0 .. 12) unroll N {
  m2[i] = m1[i] * 2;
}
```

A Predictable Stack



Modularity





Predictablity from languages to LUTs

capra.cs.cornell.edu/dahlia