Predictable Accelerator Design with Time-Sensitive Affine Types

Background:

- Specialized hardware to deal with diminishing Moore's law returns
- Reconfigurable hardware: FPGAs are hard to program
- Designed in Verilog, VHDL, Bluespec, Chisel
  RTL: Register transfer level
  Describes how data + instructions travel in the circuit at a painful detail
- Vendors offer "HLS" solutions
  $\boxed{\text{C to gates}}$ Verilog / VHDL

Comment: Verilog / VHDL is irreplaceable for $\underbrace{\text{performance / power / area}}_{\text{LUTs in FPGA}}$ tuning.

Problem with HLS tools:

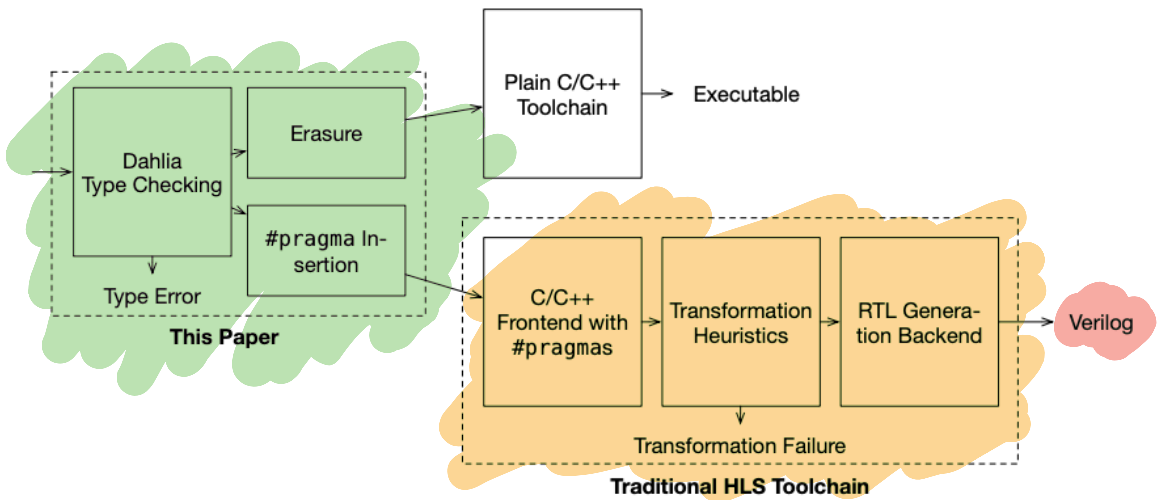
- Support only a subset of C
- Heuristics based approach to generating hardware
- Minor, Smooth changes result in large swings

$\boxed{\text{Unpredictable hardware generation}}$

Solution:

- Restrict to HLS programs with clear Hardware implementation
- Explicit annotations for "costly" implementations.

Overview:



Predictability by enforcing Subcircuits to do one thing at a time

Substructural logics are great at that!

linear, Affine

Previous works with similar idea

- Rust!
- Region based mem allocation in cyclone PLDI 2002
- A type & effect System for deterministic parallel Java: OOPSLA 200

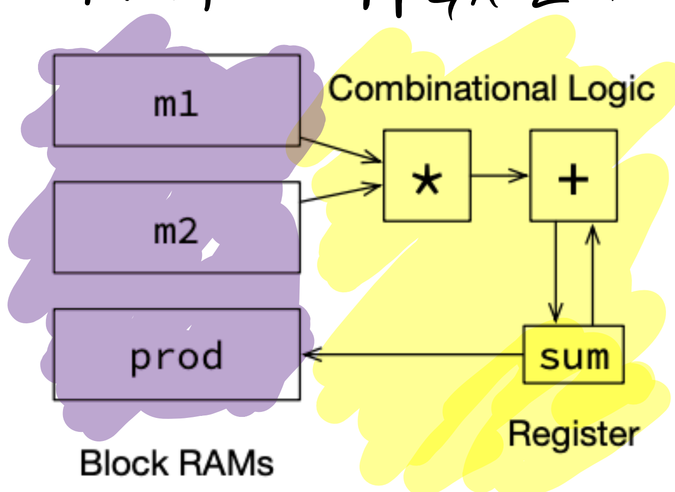
Unpredictability of HLS tools:

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

Figure 2. Dense matrix multiplication in HLS-friendly C.

Verilog would have clocks, ports and resets

RAM FPGA LUTs



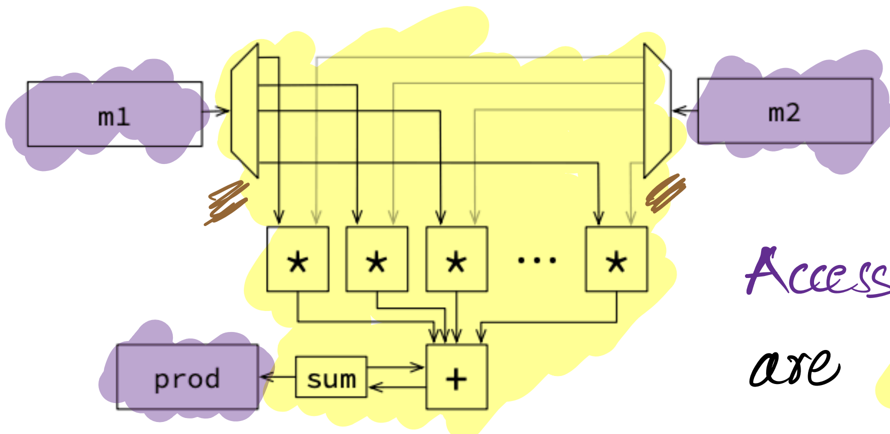
m1, m2 sit in BRAM

2.3K / 1000K LUTs used

(a) The original code.

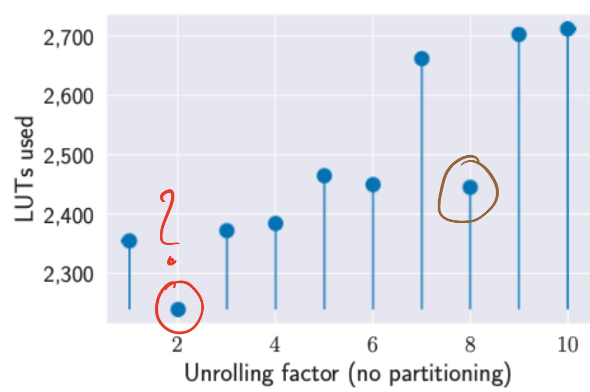
#Pragma HLS UNROLL FACTOR=8

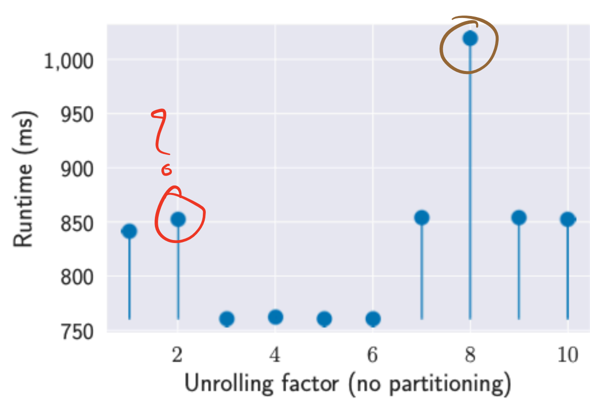
```
for (int k = 0; k < 512; k++) {
    sum += m1[i][k] * m2[k][j];
}
```



(b) With unrolling.

Accesses to mem are multiplexed



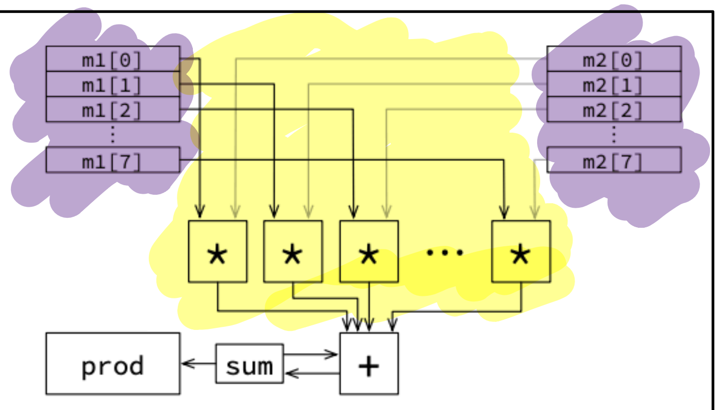


Muxing is the bottleneck.

\# HLS ARRAY_PARTITION VAR=m_1 FACTOR=8

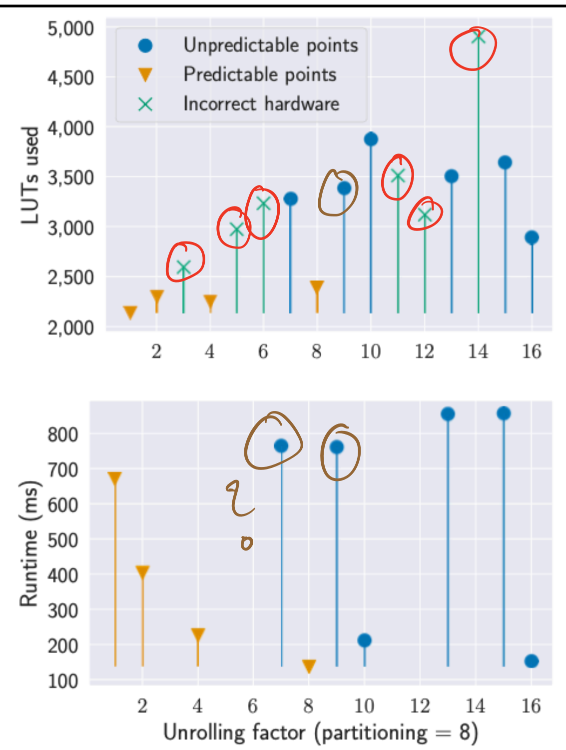
\# HLS ARRAY_PARTITION VAR=m_2 FACTOR=8

Vars split across BRAMs



(c) With unrolling and banking.

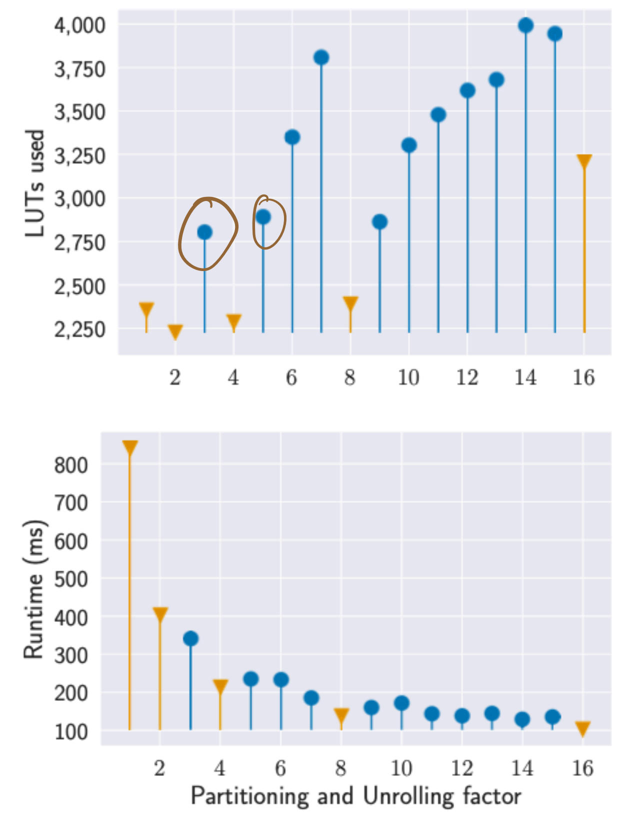
$$
\begin{aligned}
m[0] &\rightarrow bank[0] \\
m[1] &\rightarrow bank[1] \\
&\vdots \\
m[7] &\rightarrow bank[7] \\
m[8] &\rightarrow bank[0]
\end{aligned}
$$



(b) Unrolling with 8-way partitioning.

Bank Parallelism doesn't scale monotonically!

Both unrolling and banking in lockstep



(c) Unrolling and banking in lockstep.

Good performance when bank % unroll = 0

2, 4, 8, 16

Otherwise requires complicated Muxing to access data

Ex: bank factor = 8

unroll factor = 9

Every combinatorial circuit needs to access every bank

Dahlia language:

Safety property imposed

Simultaneous accesses to a given bank may not exceed the number of ports.

```
let A: float[10];
```

// memories defined by type and size.

A: BRAM in FPGA

✓
```
let x = A[5];
```
✗
```
let B = A;
```

No aliasing or duplication of mems.

✓
```
let x = A[0];    // OK
```
✗
```
A[1] := 1;       // Error:
```

The read x consumed A

✓
```
let x = A[0];
```
✓
```
let y = A[0];
```

Same address simultaneous reads are OK!

Reads are non-affine when on same index of same mem

```
let x = A[0];
let y = A[0];
```

→

let tmp = A[0];
let x = tmp;
let y = tmp;

";" **unordered composition**: statements execute in parallel

"---" **Ordered Composition**: sequential

✓

let x = A[0]

A[1] := 1

Read and write are sequential.

Restoration of resource access.

```
let A: float[10];  let B: float[10];
{
  let x = A[0] + 1
  ---
  B[1] := A[1] + x   // OK
};
let y = B[0];   // Error: B already consumed.
```

Sequential

Parallel

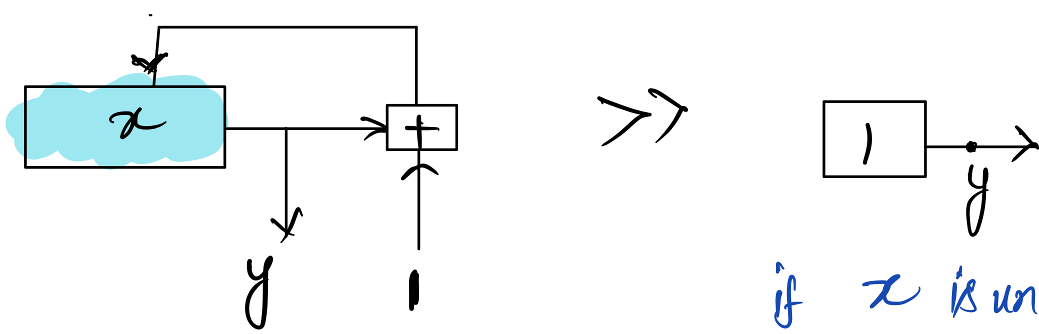
Abstract notion of time, not tied to underlying design's clock cycles.

let x = a+b ; let y = c*d

Local vars : wires & registers

```
let x = 0;   x := x + 1;   let y = x;
```

no affine typing here



if x is needed across clock cycles

if x is unused elsewhere

```
let x = A[0] + 1   ---   B[0] := A[1] + x
```

x is a register in this case.

Memory banking support

```
let A: float [n bank m];
// n % m = 0
```

Explicit affine tracking for each bank

```
let A: float[10 bank 2];
A{0}[0] := 1;
A{1}[0] := 2;   // OK: Accessing
```

float{2}[10]

Alt Syntax

let M: float{2}[4]{2}[4];

// [4 bank 2][4 bank 2]

2x2 = 4 banks.

	0,0		0,1	
	0	1	2	3
	4	5	6	7
	8	9	A	B
	C	D	E	F
	1,0		1,1	

Loops, unrolling:

```
for (let i = 0..10) unroll 2 { f(i) }
```

≡

```
for (let i = 0..5) {  f(2*i + 0);  f(2*i + 1)  }
```

for is parallelizeable, but no cross-iteration dependencies.

```
let A: float[10];
for (let i = 0..10) unroll 2 {
  A[i] := compute(i)   // Error: Insufficient banks.
}
```

for (let i = 0..8) unroll 4 { A[i] }

idx{0..4}

Check for 4 access resources

Interaction b/w composition & unrolling

```
let A: float[10 bank 2];
for (let i = 0..10) unroll 2 {
  let x = A[i]
  ---
  f(x, A[0]) }
```

```
for (let i = 0..5) {
  { let x0 = A[2*i]        ---   f(x0, A[0]) };
  { let x1 = A[2*i + 1]    ---   f(x1, A[0]) } }
```

Parallelization within each logical timestep

```
for (let i = 0..5) {
  { let x0 = A[2*i]; let x1 = A[2*i + 1] }
  ---
  { f(x0, A[0]);      f(x1, A[0]) } }
```

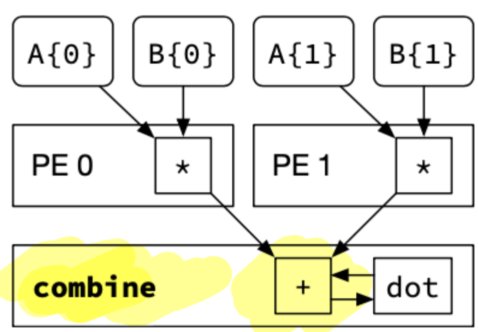
Combine construct:

```
for (let i = 0..10) unroll 2 { dot += A[i] * B[i] }
```

+= introduces silent cross iteration dependency

dot += A[2i] * B[2i]; dot += A[2i+1] * B[2i+1].

```
for (let i = 0..10)
unroll 2 {
  let v = A[i] * B[i];
} combine {
  dot += v;
}
```



Sequential code to be performed after each unrolled iteration

v is a combine register
→ tuple of all values

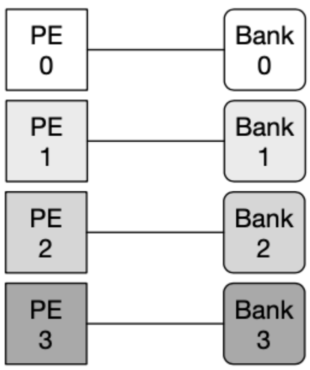
+=, -=, /=, *=

Memory views for flexible iteration:

Arbitrary computations in indices result in bad weird hardware. Ex: A[i*i]

Dahlia restricts index expr to simple expressions [i+8]

Hence: logical arrangements.

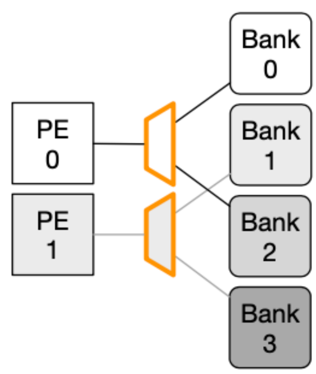


direct access

```
let A: float[8 bank 4]
for (let i = 0..8) unroll 4
  A[i];
```

Shrink:

```
let A: float[8 bank 4];
view sh = shrink A[by 2]; // sh: float[8 bank 2]
for (let i = 0..8) unroll 2
  sh[i]; // OK: sh has 2 banks. Compiled to: A[i].
```



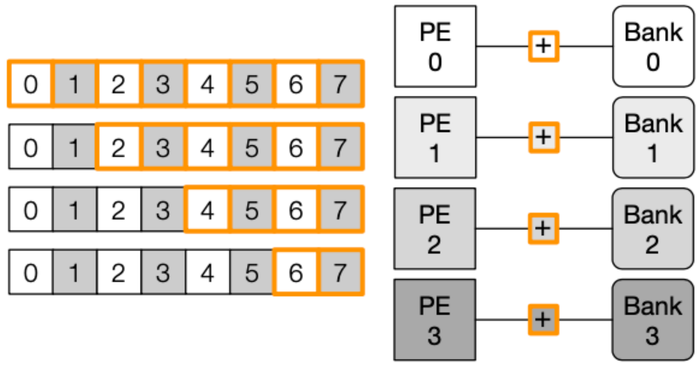
Suffix:

```
view v = suffix M[by k * e];
```

V{b}[i] = M{b}{i+e}

K = bank factor of M.

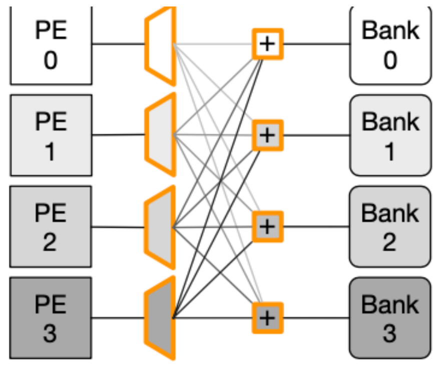
```
let A: float[8 bank 2];
for (let i = 0..4) {
  view s = suffix A[by 2*i];
  s[1]; // reads A[2*i + 1]
}
```



Shift:

```
view v = shift M[by e];
```

```
let A: float[12 bank 4];
for (let i = 0..3) {
  view r = shift A[by i*i];   // r: float[12 bank 4]
  for (let j = 0..4) unroll 4
    let x = r[j];   // accesses A[i*i + j]
}
```



Split:

For efficient h/w gen in nested loops.

```
let A, B: float[12 bank 4];
view shA, shB = shrink A[by 2], B[by 2];
for (let i = 0..6) unroll 2 {
  view vA, vB = suffix shA[by 2*i], shB[by 2*i];
  for (let j = 0..2) unroll 2 {
    let v = vA[j] + vB[j];
  } combine {
    sum += v; }}
```

Dahlia cannot reason about Seperation of V_A, V_B.

```
float A[12], B[12], sum = 0.0;
for (int i = 0; i < 6; i++)
  for (int j = 0; j < 2; j++)
    sum += A[2*i + j] * B[2*i + j];
```

view V_A = Split A [by 2];

Formalism:

$x \in$ variables $\quad a \in$ memories $\quad n \in$ numbers

b ::= **true** | **false** $\qquad v$ ::= n | b

e ::= v | **bop** e_1 e_2 | x | $a[e]$

c ::= e | **let** $x = e$ | c_1 —— c_2 | c_1 **;** c_2 | **if** x c_1 c_2 |
while x c | $x := e$ | $a[e_1] := e_2$ | **skip**

τ ::= **bit**$\langle n \rangle$ | **float** | **bool** | **mem** $\tau[n_1]$

for compiles to while

Large step semantics:

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

mem access

σ : Vars to mem map

ρ : Set of mems accessed.

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

Parallel composition

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

Sequential composition

mems accessed in c_1 are freed back (ρ_1)

Type system:

Γ: variable context (for checking index, local types)

Δ: Affine context for memories.

$$\frac{\Gamma, \Delta_1 \vdash e_1 : \mathbf{bit}\langle n \rangle \dashv \Delta_2 \qquad \Delta_2 = \Delta_3 \cup \{a \mapsto \mathbf{mem}\ \tau[n_1]\}}{\Gamma, \Delta_1 \vdash a[e] : \tau \dashv \Delta_3}$$

mem access

why the Δ_2 update for index?

$$\frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \Delta_2 \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1 \,;\, c_2 \dashv \Gamma_3, \Delta_3}$$

Parallel Composition

$$\frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \Delta_1 \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1 \mathbin{\text{——}} c_2 \dashv \Gamma_3, \Delta_2 \cap \Delta_3}$$

Seq composition

Lemma 1 (Progress). *If* $\Gamma, \Delta \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\Gamma, \Delta \sim \sigma, \rho$, *then* $\sigma, \rho, c \longrightarrow \sigma', \rho', c'$ *or* $c = \mathbf{skip}$.

Lemma 2 (Preservation). *If* $\Gamma, \Delta \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\Gamma, \Delta \sim \sigma, \rho$, *and* $\sigma, \rho, c \longrightarrow \sigma', \rho', c'$, *then* $\Gamma', \Delta' \vdash c' \dashv \Gamma_2', \Delta_2'$ *and* $\Gamma', \Delta' \sim \sigma', \rho'$.

Theorem. *If* $\emptyset, \Delta^* \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\emptyset, \emptyset, c \xrightarrow{*} \sigma, \rho, c'$ *and* $\sigma, \rho, c' \not\rightarrow$, *then* $c' = \mathbf{skip}$.

Results:

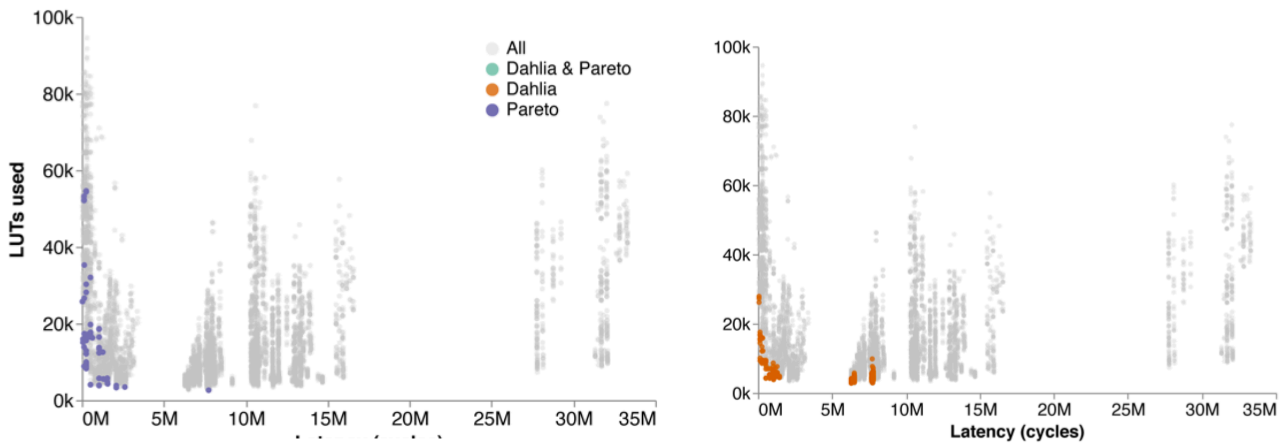
Experiments on mach HLS suite: 16/19 successful

```
for (r=0; r<row_size-2; r++)
  for (c=0; c<col_size-2; 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];
```

C code

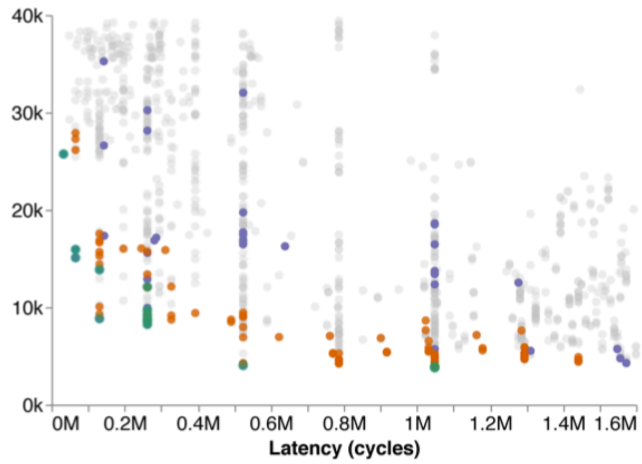
```
for (let row = 0..126) {
  for (let col = 0..62) {
    view window = shift orig[by row][by col];
    for (let k1 = 0..3) unroll 3 {
      for (let k2 = 0..3) unroll 3 {
        let mul = filter[k1][k2] * window[k1][k2];
```

Dahlia code



latency

On a macro level the programs accepted are near "optimal".



However not the exact points that traditional tools & dahlia accept.

Things that went right:

1. Deterministic design generation
2. Abstract view of time*
3. Substructural logics in H/w, an interesting idea

Things that went wrong:

1. C-like paradigm might not be the right mental model for h/w in general.
2. The hardware level optimisations manifest as weird syntax
3. Unlike s/w langs, user needs to inspect the generated h/w and that ability isn't clear in general with HLS

4. verilog has generate & for that behave similarly !!