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
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
Ada’s Journey
Ada’s Journey
Ada’s Journey
Ada’s Journey
Hardware Design

FPGA
Hardware Design

FPGA

Traditional HDLs

SystemVerilog
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
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

- Time is real
- Concurrent semantics

```systemverilog
time module neptune_metadata_unpack #(  .TUSER_W (TUSER_W))
)

  // 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 ()

);
```
Traditional HDLs

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

```systemverilog
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 ()
);
```
Hardware Design

Traditional HDLs

FPGA
Hardware Design

SystemVerilog

Traditional HDLs

Modern HDLs

FPGA
// 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))

}
// 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))

}
Modern HDLs

```scala
// 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
FPGA Hardware Design

Traditional HDLs

Modern HDLs

SystemVerilog

FPGA
int m1[512], m2[512], sum;
for (int i = 0; i < 512; i++) {
    sum += m1[i] * m2[i]
}
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
loops
arrays
break
malloc
functions
conditionals
method calls

loops
arrays
break
malloc
functions
conditionals
method calls
arrays
malloc
loops
method calls
arrays
break
malloc
functions*
conditionals
High-performance hardware designs

- loops
- arrays
- break
- malloc
- functions
- conditionals

*These keywords may not be directly relevant to the image content.
Ada’s Journey
Super secret™ accelerator

Hardware
Super secret™ accelerator

```c
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;
    }
}
```
Super secret™ accelerator

```c
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;
    }
}
```
Super secret™ accelerator

```c
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

```
  m1
   * 
    +
      sum
```

Block RAM

Register
Super secret™ accelerator

```c
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

```
+  
*   
```

Block RAM  

Register  

Super secret™ accelerator
Super secret™ accelerator

```c
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;
    }
}
```
Super secret™ accelerator

```c
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

```
<table>
<thead>
<tr>
<th>m1</th>
<th>*</th>
<th>m2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

Block RAM

```
<table>
<thead>
<tr>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Register
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
Resources on the FPGA

Super secret™
accelerator

- Area
- Latency vs. Parallelism

- Area vs. Parallelism
Area-Latency trade-offs

Create more processing elements to extract parallelism
Super secret™ accelerator

```c
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;
    }
}
```
Super secret™ accelerator

```c
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;
    }
}
```
Super secret™ accelerator

```c
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;
    }
}
```
Area-Latency trade-offs
Area-Latency trade-offs
Hardware

Diagram:

- m1
- m2
- prod

* * *

+ sum

Mathematical expression:

\[ \text{sum} = (m_1 \times (m_2 \times \text{prod})) \]
Hardware

```
ml * m2 * prod * + sum
```
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)"

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

Unable to schedule 'load' operation
Super secret™ accelerator

```c
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;
    }
}
```
Super secret™ accelerator

```c
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

Logical Memory
Generated hardware

Memory Partitions

\[ \text{prod} \times \text{m}^2 \times \text{prod} \rightarrow \sum \]
Super secret™ accelerator

```c
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;
    }
}
```
Super secret™ accelerator

```c
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

512 \% 3 \neq 0
Mismatched Partition Sizes

\[ 512 \mod 3 \neq 0 \]
Hardware

Mismatched Partition Sizes
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!

Secrets of HLS
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 \rightarrow set of reads or writes to m
3 G = \emptyset set of sets of compatible accesses
4 for all accesses a in A:
5     for all sets of accesses g in G:
6         if Comp(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 Function ConfigureMemory:
13 input: A \rightarrow set of reads of m
14 input: A_r \rightarrow set of writes to m
15 G_r = GroupAccesses(A_r)
16 G_m = GroupAccesses(A_m)
17 I = \emptyset set of memory instances
18 for all read sets R in G_r:
19     I_r = \{R\}
20     i_r = ReachingWrites(G_r, I_r)
21     i = BankAndBuffer(i_r, I_r)
22     for each inst in i:
23         i_r' = ReadSets(inst) + R
24     if Comp(A_1, A_2) \forall A_1, A_2 \in (G_m \cup I_r') then:
25         i_r = BankAndBuffer(i_r', I_r)
26     if Cost(i') < Cost(i) + Cost(inst) then:
27         remove inst from I
28     add i' to I
29     break
30     if i has not been merged then add i to I
31 return I
32 end function
```

Figure 7. Banking and buffering algorithm for calculating instances of on-chip memory m.
Spatial: A Language and Compiler for Application Accelerators

- DSP used
- BRAM used
- LUT used

Unrolling Factor

Normalized Resource Usages
Ada’s Journey
Dahlia
Predictable Accelerator Design
let m1: float[10];

let x = m1[0];
m1[1] := 1;
let m1: float[10];

let x = m1[0];
m1[1] := 1;
let m1: float[10];

let x = m1[0];
m1[1] := 1;
let m1: float[10];

let x = m1[0];

m1[1] := 1;

Do these run concurrently?
let m1: float[10];

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

Do these run concurrently?
let m1: float[10];

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

Do these run concurrently?

Error: Affine resource ‘m1’ already used in this context.
let m1: float[10];

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

let m1: float[10];

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

At most one use of a variable
Dahlia

**Affine Type System**

**Declaration**

\[
\text{let } m_1: \text{float}[10];
\]

**Use**

\[
\text{let } x = m_1[0]; \\
\text{m}_1[1] := 1;
\]

**Error:** Affine resource \('m_1'\) already used in this context.

**At most one use of a variable**
Dahlia

```haskell
let m1: float[10];
let x = m1[0];
m1[1] := 1;
```

Do these run concurrently?
let m1: float[10];

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

Do these run concurrently?

Unordered Composition
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.
let m1: float[10];

let x = m1[0];

m1[1] := 1;
let m1: float[10];

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

Ordered Composition
let m1: float[10];

let x = m1[0];

---

m1[1] := 1;

**Ordered Composition**

- Run sequentially.
- Renew affine resources.
Temporally exclusive use of resources
Temporally exclusive use of resources
\{ A \quad \cdash \quad B \};

C

**Temporally exclusive** use of resources
let m1: float[12];
let m2: float[12];

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

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

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

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

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

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

Dahlia

Hardware
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
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]
    }
}
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
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
Language Features
Language Features

Formalism

\[
x \in \text{variables} \quad \alpha \in \text{memories} \quad n \in \text{numbers}
\]

\[
b ::= \text{true} | \text{false} \quad \nu ::= n | b
\]

\[
e ::= \nu | \text{bop} \ c_1 \ c_2 \ | \ x \ [a|e]\n\]

\[
c ::= e \ | \ \text{let} \ x = e \ | \ c_1 \ ::= c_2 \ | \ c_1 ; c_2 \ | \ \text{if} \ x \ c_1 \ c_2 \ |
\]

\[
\text{while} \ e \ [x = e \ [a|e] \ ::= c_2 \ | \ \text{skip}
\]

\[
r ::= \text{bit}(n) | \text{float} | \text{bool} | \text{mem} \ \{\nu\}
\]
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

32,000 designs
2,600 hours of estimation

Blocked matrix-matrix multiply
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

```c
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

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
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
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]
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]
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
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]
The Future
let m1: float[12 bank M_1];
let m2: float[12 bank M_2];

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

```javascript
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];
}
```
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

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**

```python
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

- High-Level Languages
- Hardware Description Languages
- Netlist
- Hardware
The Future

A Predictable Stack

High-Level Languages

Hardware Description Languages

Netlist

Hardware
The Future

A Predictable Stack

- High-Level Languages
- Hardware Description Languages
- Netlist
- Hardware
The Future

A **Predictable** Stack

- High-Level Languages
- Hardware Description Languages
- Netlist
- Hardware

???
The Future

Resource Polymorphism

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

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

Modularity

A Predictable Stack

Predictability from languages to LUTs

capra.cs.cornell.edu/dahlia