Connecting Polyhedral Optimization to CGRA Buffer Generation

Dillon Huff and Qiaoyi Liu
Hardware-Software Compiler Interaction

Hardware Compilers
- PEAk Program
- Lake Program
- Canal Program

Low-Level DSLs
- Fault (HVL)
- Magma (HDL)
- CoreIR

Rewrite Rules
- PEAk: PE Generator
- Lake: Memory Generator
- Canal: Interconnect Generator
- CoreIR

Routing Graph
- CGRA Verilog

Software Compiler
- Halide
- Lower
- CoreIR
- Map PE and Memory
- Mapped CoreIR
- CGRA Place & Route
- CGRA Bitstream

CPU Code
Unified buffers: a hardware independent memory abstraction

*Halide2Hardware*

- Define applications
- Define Accelerator domain and memory hierarchy

*Loop Nest*

- It defines how data flow between statements (load and store)

*Unified Buffer*

- Elaborate into Polyhedral IR
  - Iteration domain
  - Access map
  - Schedule

Backend Memory Constraints

*Hardware Rewrites*

- Memory Tile Configuration
- CoreIR Topology Graph
Unified buffers: a hardware independent memory abstraction

Halide2Hardware

Halide
- Define applications
- Define Accelerator domain and memory hierarchy

Loop Nest
- It defines how data flow between statements (load and store)

Unified Buffer
- Elaborate into Polyhedral IR
  - Iteration domain
  - Access map
  - Schedule

Buffer Extraction

Hardware Rewrites

Backend Memory Constraints

Memory Tile Configuration + CoreIR Topology Graph
What comes out of the Halide front end:

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4
This application has 2 stages: brighten and blur

```python
for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4
```

Read in a 64 x 64 input and brighten each pixel.
This application has 2 stages: brighten and blur

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4

Produce a blurred version of the bright image by averaging together 2x2 squares
Compute mapping creates processing elements (PEs) for each stage

for r in [0, 63]:
    for c in [0, 63]:
        \[ br[r, c] = 2 \times \text{in}[r, c] \]

for r in [0, 62]:
    for c in [0, 62]:
        \[ \text{out}[c, r] = \frac{(br[r, c] + br[r + 1, c] + br[r, c + 1] + br[r + 1, c + 1])}{4} \]
But how do these PEs communicate?

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4
Through Memory

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4
Problem 1: Optimizing memory capacity

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
            br[r, c + 1] + br[r + 1, c + 1]) / 4

64 x 64 if program runs sequentially
Solution: Execute the loop nests in parallel as separate tasks

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4

But only 64 + 2 entries are needed if stages run in parallel
Problem 2: Bandwidth demands for most PEs are much higher than an SRAM can provide

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4
1 input / cycle and 4 outputs / cycle

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
            br[r, c + 1] + br[r + 1, c + 1]) / 4
Typical solution in industrial hardware compilers: Give up and reduce throughput

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
            br[r, c + 1] + br[r + 1, c + 1]) / 4
Typical solution in industrial hardware compilers: Give up and reduce throughput

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
            br[r, c + 1] + br[r + 1, c + 1]) / 4
Typical solution in industrial hardware compilers: Give up and reduce throughput

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4
Typical solution in industrial hardware compilers: Give up and reduce throughput

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
            br[r, c + 1] + br[r + 1, c + 1]) / 4
Typical solution in industrial hardware compilers: Give up and reduce throughput

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4
Typical solution in industrial hardware compilers: Give up and reduce throughput

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4
Our solution: Use polyhedral analysis to generate a synthesizable, high bandwidth implementation

for $r$ in $[0, 63]$:
    for $c$ in $[0, 63]$:
        $br[r, c] = 2 \times in[r, c]$

for $r$ in $[0, 62]$:
    for $c$ in $[0, 62]$:
        $out[c, r] =$
            $(br[r, c] + br[r + 1, c] +$
            $br[r, c + 1] + br[r + 1, c + 1]) / 4$
But this is easier said than done...

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4
Important restriction: All memory access expressions are affine

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
        (br[r, c] + br[r + 1, c] +
        br[r, c + 1] + br[r + 1, c + 1]) / 4
Important restriction: All memory access expressions are affine

We can use polyhedral analysis to design our memory optimizations!
And many components of this problem have already been formulated in the polyhedral model

- Re-scheduling operations for higher locality
- Checking the legality of memory banking schemes
- Performing storage folding
- Introducing multi-level re-use buffers
But mostly in other contexts...

- Software optimization
- HLS targeting FPGA / ASIC technology libraries with fine grained control and memory primitives (gates, LUTs, SRAM macros)
How do we transform a high bandwidth buffer with a fixed, statically analyzable access pattern into hardware that can be implemented on our CGRA?
Unified buffer: a hardware independent memory abstraction

**Halide2Hardware**
- Define applications
- Define Accelerator domain and memory hierarchy

**Loop Nest**
- It defines how data flow between statements (load and store)

**Unified Buffer**
- Elaborate into Polyhedral IR
  - Iteration domain
  - Access map
  - Schedule

**Hardware Rewrites**
- Memory Tile Configuration
- CoreIR Topology Graph

**Backend Memory Constraints**
An Application Memory

for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4

Buffer for br

1 input port
4 output ports
for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]

for r in [0, 62]:
    for c in [0, 62]:
        out[c, r] =
            (br[r, c] + br[r + 1, c] +
             br[r, c + 1] + br[r + 1, c + 1]) / 4
Exhaustive banking: intersect the map

- Naïve banking
  - Dual port sram
- Input -> output
  - Intersect the range

Buffer for br

1 input port

4 output ports

Switch

B 0
B 1
B 2
B 3
Statically analyze the channel dependency
Fuse the loop

for r in [0, 63]:
    for c in [0, 63]:
        \(br[r, c] = 2 \times \text{in}[r, c]\)

for r in [0, 62]:
    for c in [0, 62]:
        \(\text{out}[c, r] = \frac{1}{4}(br[r, c] + br[r + 1, c] + br[r, c + 1] + br[r + 1, c + 1]) / 4\)

for r in [0, 63]:
    for c in [0, 63]:
        \(br[r, c] = 2 \times \text{in}[r, c]\)

if (r>0 && c>0)
    \(\text{out}[c, r] = \frac{1}{4}(br[r, c] + br[r, c + 1] + br[r + 1, c] + br[r + 1, c + 1])\).
Compute Dependence Distance (DD)

- Compute dependence distance after loop fusion
  - For each bank, calculate the write has been made between the oldest and latest data
- It’s constant for all the 2x2 blur UBuffer banks

```python
for r in [0, 63]:
    for c in [0, 63]:
        br[r, c] = 2*in[r, c]
    if (r>0 && c>0):
        out[c, r] =
        ¼*(br[r, c] + //DD = 0
        br[r , c + 1] + //DD = 1
        br[r + 1, c ] + //DD = 64
        br[r + 1, c + 1]). //DD = 65
```

1 input port
4 output ports
Naïve Bank Merging

1 input port

switch

B0, B1, B2, B3

4 output ports

Depth=1

1 input port

Depth=63

Depth=1

4 output ports
Naïve Bank Merging

- Already address the two problems
- Still backend independent, non-optimal for the CGRA
Backend Aware: Local Reuse / CGRA Routing
Backend Aware: Memory Tile Interface

Memory backend:
1 in port
1 out port

Memory backend:
2 in ports
2 out ports

Dangling wire

Dangling wire
Backend Aware: Memory Tile Interface (3x3 blur)

Memory backend:  
1 in port  
1 out port

Memory backend:  
2 in ports  
2 out ports  
Dangling wire
Split the Memtile: Chaining

- **Memory Tile Internal Constraint**
  - Capacity = 1KB
  - Decouple into multiple tiles if exceed

![Diagram of HW Mem and Chain in](image-url)
Look inside the Memtile: Wider fetch width

- Memory Tile Internal Constraint
  - Fetch width = 4

- Compiler Transformation: Vectorization and split loop nest
  - 2 loop nest & 1 buffer between
  - 4 loop nests & 3 buffers in between
More detail will be covered in the Memory deep dive talk
Conclusion

- Accelerator push memory (Unified Buffer) can be modeled as dataflow channels between read and write.

- Polyhedral analysis
  - Optimized capacity
  - Fulfill bandwidth requirements

- In order to target a specific hardware backend, backend specific rewrite rules are proposed.