Metal FS: File System and Data Stream Operators for FPGAs

Robert Schmid, Lukas Wenzel

Operating Systems and Middleware Group

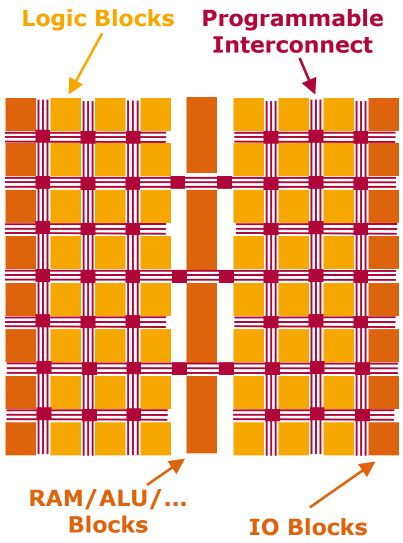
30.01.2019

Introduction: What is an FPGA?

- **F**ield-**P**rogrammable **G**ate **A**rray: programmable hardware circuit
- Algorithms are represented as a hardware description
- Programming:
  - VHDL (Hardware Description Language)
  - High-Level Synthesis (HLS)

- Interest in FPGAs is growing (again)
  - Energy efficiency
  - Parallel and pipelined data processing
  - ‘Computing wires’

- My focus: FPGAs as part of a computer system



Development of Heterogenous Computing

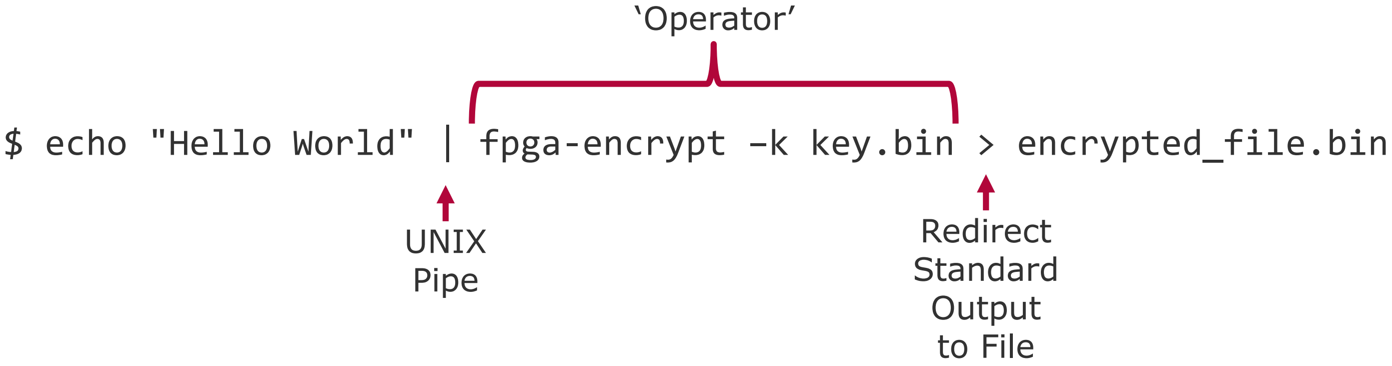
- Physical limitations in general-purpose processor development
  - Energy efficiency becomes a key question

- FPGAs and specialized co-processors are suited for a variety of use cases
  - Google TPU (TensorFlow)
  - Microsoft HPU (HoloLens)
  - Apple T2 (MacBook, iMac Encryption)
  - Cloud FPGAs: Amazon F1, Azure ML Service

- Innovations in interconnects and system architecture
  - “Accelerators become first-class citizens in the system”

First-class citizens?

- How should end-users interact with FPGAs?
- Just like with any other executable program!
- Analogy: Builtin UNIX tools (cat, grep, sed, awk, …)
  - Do one thing, and do it well!



‘Operator’

```
$ echo "Hello World" | fpga-encrypt -k key.bin > encrypted_file.bin
```

UNIX Pipe

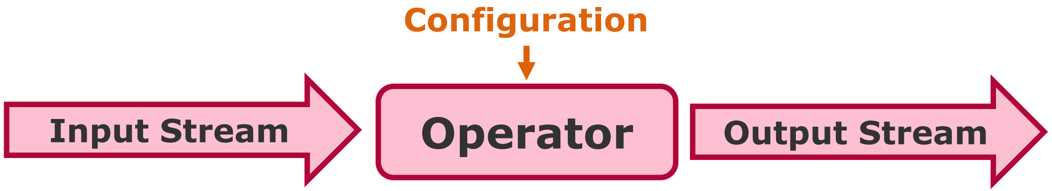
Redirect Standard Output to File

Metal FS

- Started as part of a File System project seminar
- Goal: Improve the accessibility of FPGA accelerators using a file system abstraction

Foundations IBM POWER CAPI + SNAP Xilinx Vivado

Operators are specified in Vivado HLS



```
static snapu64_t _offset = 0;

mtl_retc_t my_metal_operator_set_offset(snapu64_t offset) {
    _offset = offset;
    return SNAP_RETC_SUCCESS;
}

void my_metal_operator(mtl_stream & in, mtl_stream & out) {
    mtl_stream_element element;
    do {
        element = in.read();
        element.data += _offset;
        out.write(element);
    } while (!element.last);
}
```

Chaining Operators

- What happens here?

```
$ cat encrypted_file.bin | fpga-decrypt | fpga-uppercase
HELLO WORLD
```

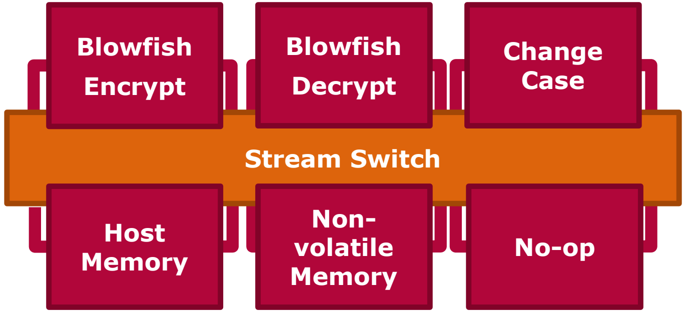
- In between FPGA processing steps, data should not be copied to the CPU's main memory (slow)

- In conclusion:
  - Multiple operators must be deployed on the FPGA at once
  - Active subset and order of Operators should be configurable at runtime

Metal FS Operator Pipelines

- Streaming Data from different types of memory

- Composition of Pipelines by using AXI Stream Switch



Blowfish Encrypt

Blowfish Decrypt

Change Case

Stream Switch

Host Memory

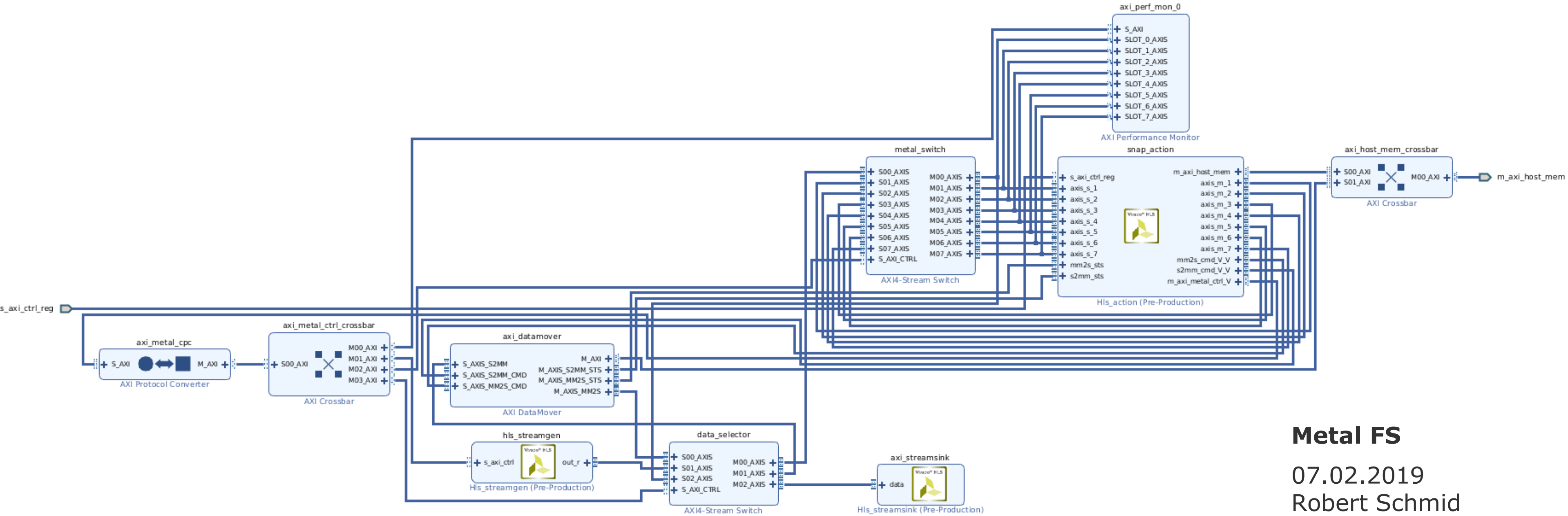
Non-volatile Memory

No-op

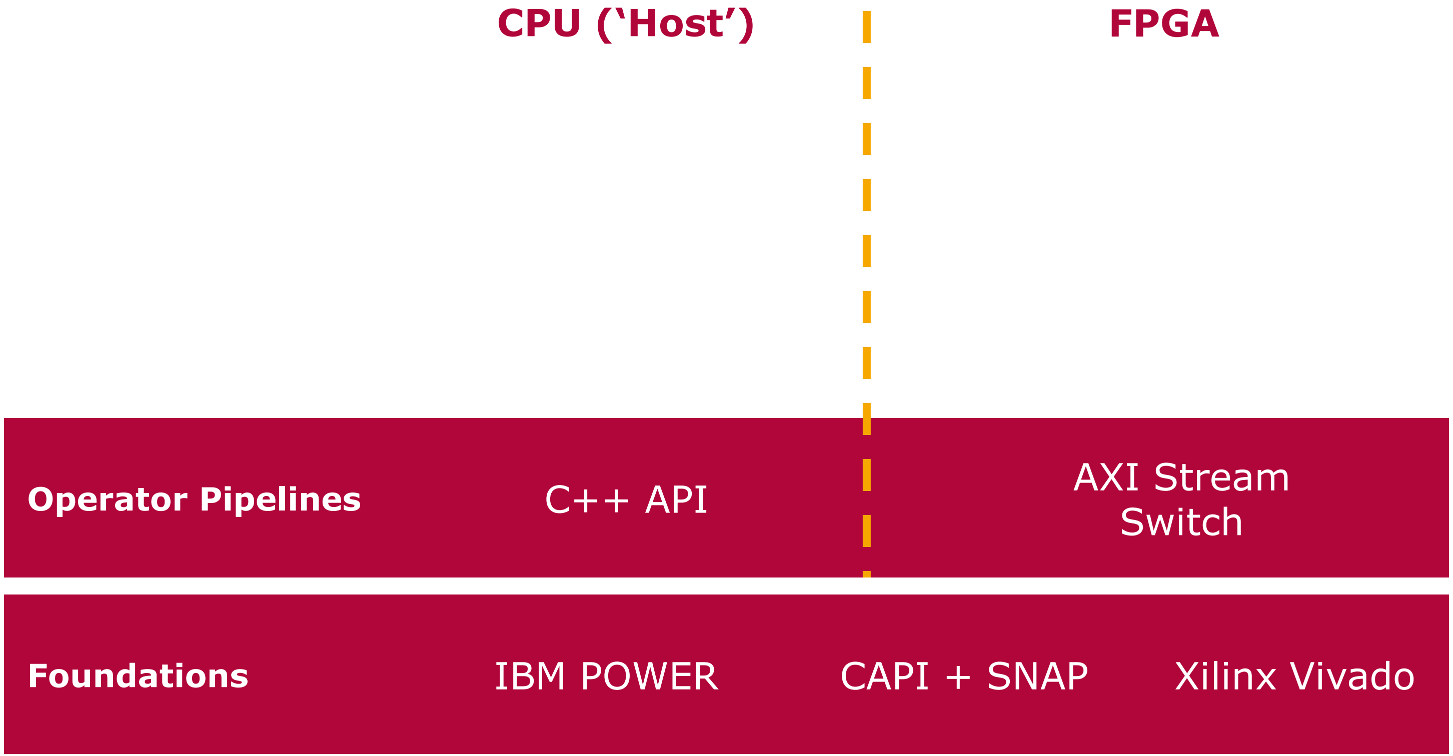
- C++ API

```
OperatorRegistry registry;

auto encrypt = registry.operators().at("encrypt");
encrypt->setOption("key", keyBuffer);

auto dataSource = create_data_source(inputBuffer);
auto dataSink = create_data_sink(outputBuffer);

PipelineDefinition pipeline
    ({ dataSource, encrypt, dataSink });
pipeline.run();
```

Composition of Hardware Components: Block Design



Metal FS: Architecture Overview



CPU (‘Host’)

FPGA

Operator Pipelines

C++ API

AXI Stream Switch

Foundations

IBM POWER

CAPI + SNAP

Xilinx Vivado

Metal FS Hybrid Filesystem

- Leverage the NVMe storage on the Nallatech N250S FPGA card
- One use case for Operator Pipelines

- File System Metadata is maintained in an LMDB Key-Value Store on the host
    - inodes, directory entries, free extents
- Block Mapper on the FPGA translates file offsets to physical addresses using extent lists
- All file accesses are implemented as Operator Pipelines (Read -> Write)
    - Data transformations can be transparently added (e.g. encryption)

Metal FS: Architecture Overview

	CPU (‘Host’)	FPGA	
Hybrid File System	Filesystem Metadata Store	Data Sources and Sinks	Block Mapper
Operator Pipelines	C++ API	AXI Stream Switch	
Foundations	IBM POWER	CAPI + SNAP	Xilinx Vivado

Metal FS FUSE Filesystem

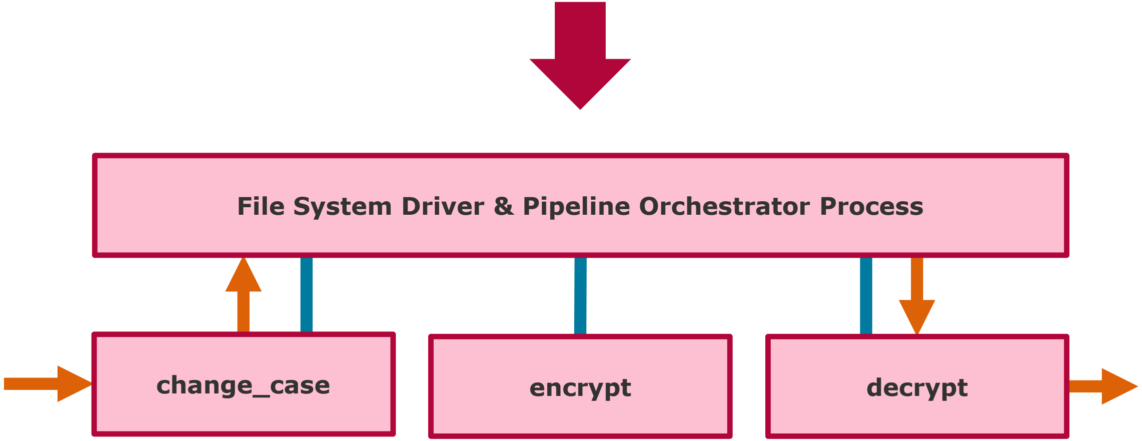
- Users can mount Metal FS as a Linux file system
- Implemented in user space

- Example:

```
$ cp ~/orders.tbl /metal_fs/files/
```

Metal FS Symbolic Executables

```
$ echo "Hello World" \
    | /metal_fs/operators/change_case \
        | /metal_fs/operators/encrypt \
            | /metal_fs/operators/decrypt
```

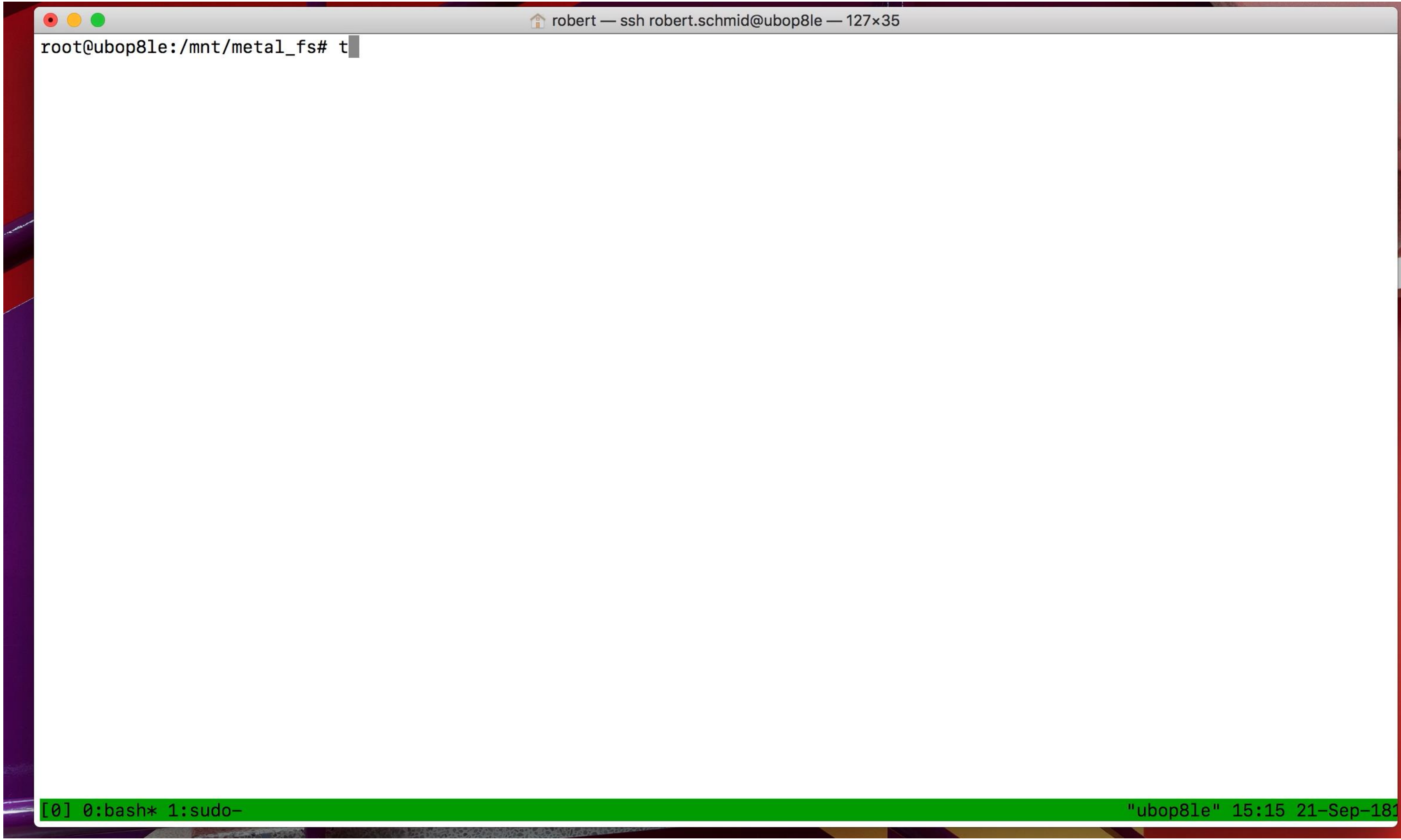


🟦 Message flow via UNIX Socket

🟧 Data flow via Memory-Mapped Files

Metal FS: Architecture Overview

	CPU (‘Host’)	**FPGA**
User Interface & Instrumentation	Linux Filesystem Driver Symbolic Executables	AXI Performance Monitor
Hybrid File System	Filesystem Metadata Store	Data Sources and Sinks Block Mapper
Operator Pipelines	C++ API	AXI Stream Switch
Foundations	IBM POWER	CAPI + SNAP Xilinx Vivado



```
root@ubop8le:/mnt/metal_fs# t
```

```
[0] 0:bash* 1:sudo-                                                                          "ubop8le" 15:15 21-Sep-18
```

Hands-On: Set up Metal FS Simulation

- All terminal commands are listed in `simulation_cmd.md`

- Steps:
  1. Clone Metal FS: `https://github.com/osmhpi/metal_fs`
  2. Compile Metal FS Software
  3. Build Hardware Simulation Image through SNAP
  4. Start Simulation, mount Metal FS and run the change_case operator

Anatomy of an Operator in Vivado HLS

- HLS translates the `mtl_stream` references into AXI Stream interfaces
- We require the `keep` and `last` signals which are optional channels in the AXI Stream protocol

```
struct mtl_stream_element {
  ap_uint<64> data; ap_uint<8> keep; ap_uint<1> last;
};

void my_operator(mtl_stream &in, mtl_stream &out) {
    mtl_stream_element element;
   do {
       element = in.read();
       out.write(element);
   } while (!element.last);
}
```

Programming with HLS: Arbitrary-precision integers

- HLS offers the `ap_uint` types for integers with arbitrary bit precision
- `snapu{8, 16, 32, 64}_t` are typedefs for `ap_uint<>`
- Access bit ranges of an `ap_uint` like this (similar to VHDL):

```
snapu16_t my_integer;
snapu8_t high_byte = my_integer(15, 8);
```

- Concatenate integers:

```
snapu8_t high_byte = 0xFF;
snapu8_t low_byte  = 0x0A;
snapu16_t both_bytes = (high_byte, low_byte);
```

Hands-On: Build a Grayscale Filter operator

- Goal: Implement an operator that processes a bitmap image and converts it to grayscale, except for pixels where red is the dominant color

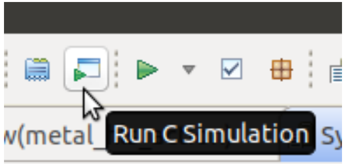
- An operator skeleton is prepared in `metal_fs/src/metal_fpga/hw/hls/hls_action/mtl_op_image.cpp`

- Bitmap header is not aligned to an 8-byte boundary
  - The skeleton temporarily inserts a padding to make processing easier

- The Vivado HLS IDE provides debugging and analysis capabilities

Hands-On: Build a Grayscale Filter operator

- Steps
  1. Implement the grayscale conversion algorithm in the `process_stream` method
     - Use the Vivado HLS IDE
  2. Use the HLS Testbench to check if your algorithm works when executed as software



  3. Run SNAP’s `make model` to translate it into a hardware description
  4. Try it out (with a scaled down image) in the simulation environment

```
/metal_fs/src/metal_fpga/hw/hls/hls_action/apples_simulation.bmp
```

  5. Run your operator in profiling mode, to see how well it performs

```
/operators/image -p
```

Performance Counters in Metal FS

- Performance Counters for Input and Output interfaces of an Operator
    - Bytes transferred
    - Active cycles
    - Idle cycles (caused by Producer/Consumer)
    - Total cycles

- Usage

```
cat apples.bmp | /mnt/metal_fs/operators/image -p > applesbw.bmp

STREAM  BYTES TRANSFERRED  ACTIVE CYCLES  DATA WAIT      CONSUMER WAIT  TOTAL CYCLES  MB/s
input   6538               818       59%  568       41%  0          0%  1389          1176.75
output  6538               818       59%  568       41%  0          0%  1389          1176.75
```

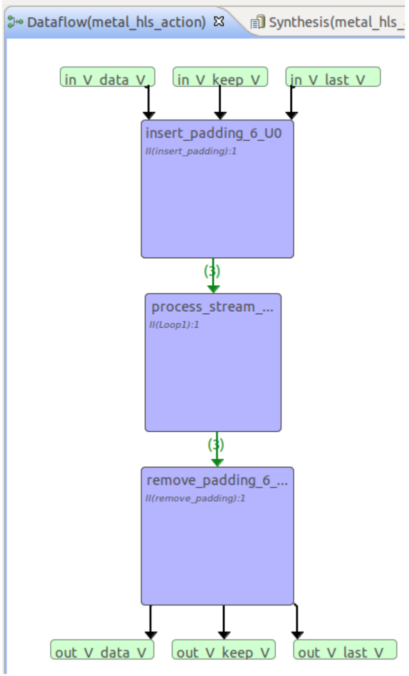
Performance Optimization in Vivado HLS

Use the HLS Pipeline pragma to reduce the 'initiation interval' of your loop iterations

```
#pragma HLS PIPELINE
```

- Watch out for warnings in the HLS synthesis compiler log

DataFlow Analysis View:



Hands-On: Optimize the operator performance

- Steps:
  1. Check if your operator is the bottleneck by using Metal FS profiling output

```
cat apples.bmp | /mnt/metal_fs/operators/image -p > applesbw.bmp
STREAM  BYTES TRANSFERRED  ACTIVE CYCLES  DATA WAIT       CONSUMER WAIT  TOTAL CYCLES  MB/s
input   6538               818        22%  502         13%  2403       64%  3729          438.32
output  6538               818        22%  2908        78%  0           0%  3729          438.32
```

  2. Explore the Analysis View in Vivado HLS
  3. Try out the HLS PIPELINE pragma

Thank you
for your attention!