HEAD

HardwarE Accelerated Deduplication

Final Report

CS710 Computing Acceleration with FPGA

December 9, 2016

Insu Jang

Seikwon Kim

Seonyoung Lee

Executive Summary

- ✓ A-Z development of deduplication
  - ✓ SW version of deduplication
  - ✓ Chunking HW logic
  - ✓ Fingerprinting HW logic
  - ✓ HW device driver on Petalinux
  - ✓ Merge deduplication SW-HW

- ✓ Performance Benefit
  - ✓ ARM+FPGA
    - ✓ 8x faster than ARM only
    - ✓ 3x faster than x86_64
  - ✓ With low power consumption

Background

Deduplication Concept

- Data compression technique
  - Eliminate duplicate copies
    - Reduce overall cost of storage
    - Manage data growth

Deduplication Concept

- Cloud storage **without deduplication**

Deduplication Concept

- Cloud storage **with deduplication**

Deduplication Process

- File chunking
  - Fixed size chunking
  - Variable size chunking
- Chunk finger printing
- Eliminating duplicates

File Chunking

- Divide file into chunks
- Chunk
  - Portion of data, presumed to be duplicate
  - Parts that will reform a file

This is a chunk__

This might be a same chunk__

This is CS710__

This is a chunk__

Fixed Size Chunking

≠

Fixed Size Chunking

5 Byte

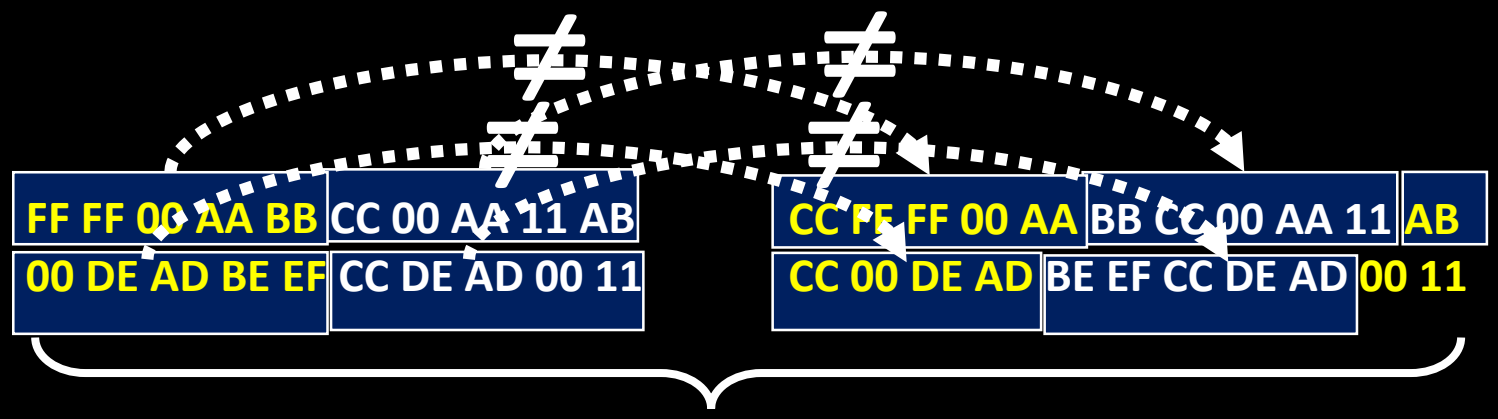
FF FF 00 AA BB CC 00 AA 11 AB
00 DE AD BE EF CC DE AD 00 11

5 Byte

CC FF FF 00 AA BB CC 00 AA 11 AB
CC 00 DE AD BE EF CC DE AD 00 11

Fixed Size Chunking

- Pros
  - fast way of chunking
- Cons
  - Deduplication ratio



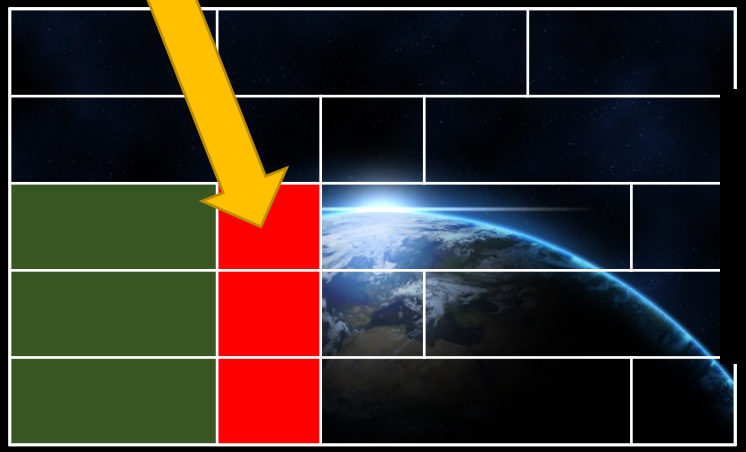
0%

Variable Size Chunking

Variable Size Chunking

Delimiter ‘CC’

FF FF 00 AA BB CC 00 AA 11 AB
00 DE AD BE EF CC DE AD 00 11

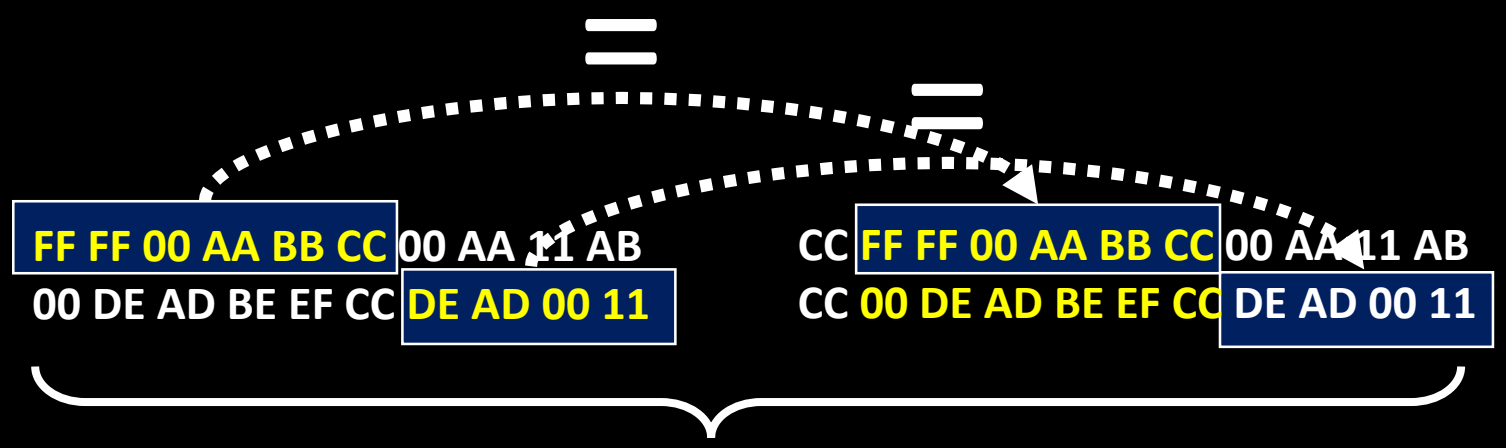


Delimiter ‘CC’

CC FF FF 00 AA BB CC 00 AA 11 AB
CC 00 DE AD BE EF CC DE AD 00 11

Variable Size Chunking

- Pros
  - Deduplication ratio
- Cons
  - Slower than fixed chunking



50%

Chunk Fingerprinting

- Transform chunks into shorter values
  - Faster to compare between chunks

13 00 14

00 BB 12

13 00 14

16 86 00

13 00 14

AA BE EF

Eliminating Duplicates

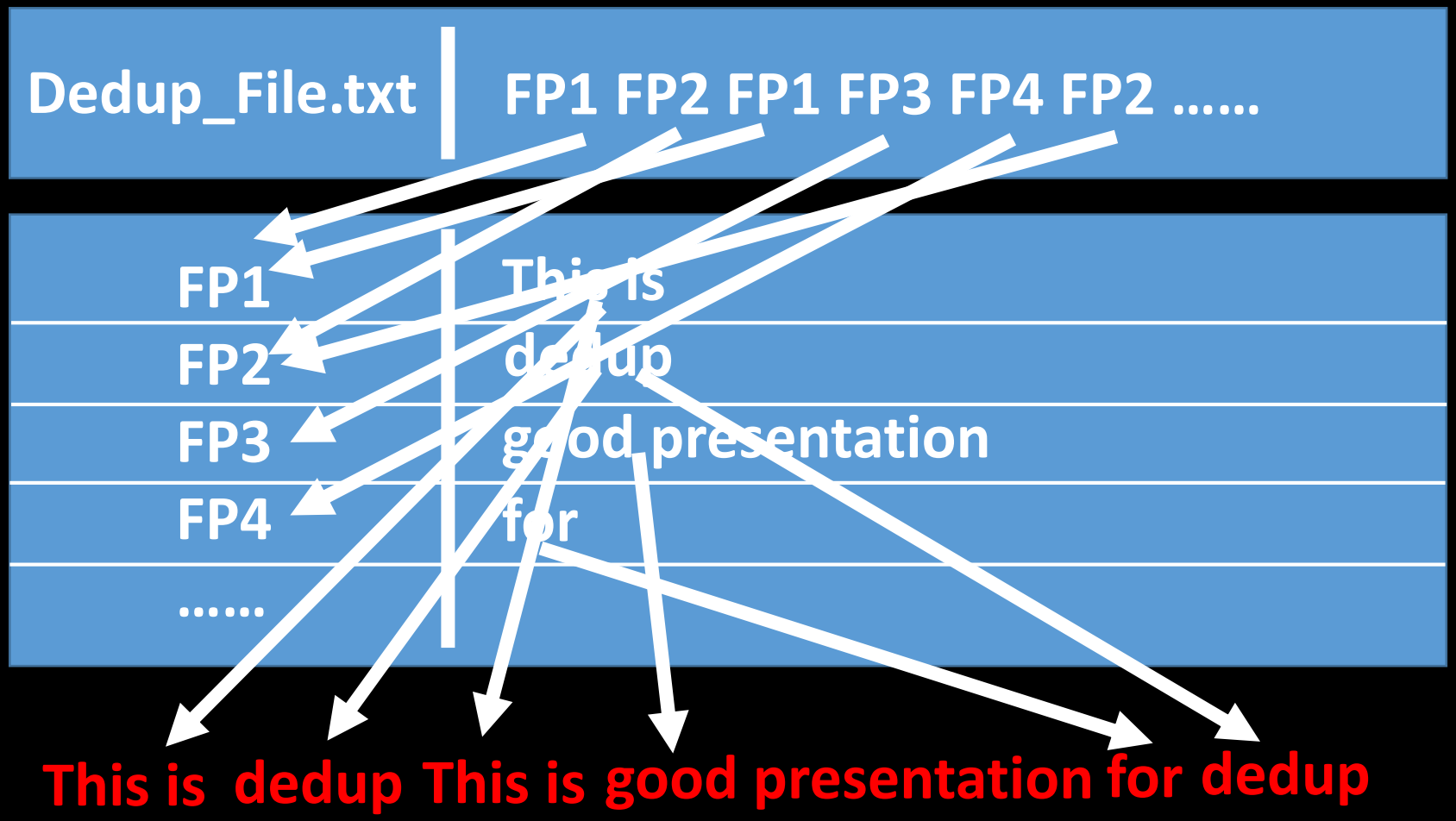
13 00 14

00 BB 12

16 86 00

AA BE EF

Restoring the Dedup File

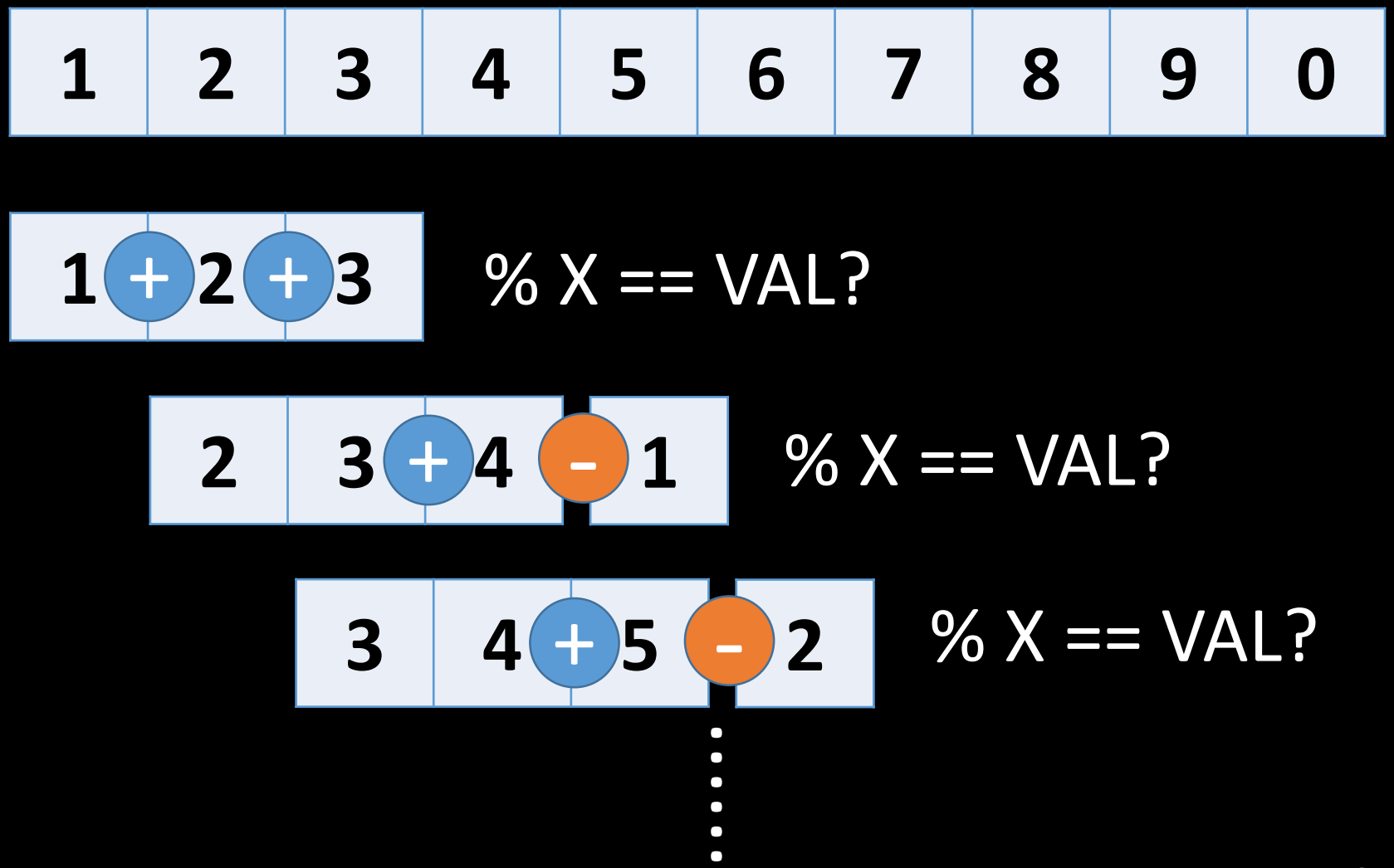


Deduplication Key Design Issues

- How to create chunks
  - Variable size chunks
  - Fast and simple rolling hash algorithm

- How to finger print
  - Collision-free hash
    - ✓ Murmur
    - SHA-256
    - FNV

Fast & Simple Rolling Hash



Finger Printing Murmur Hash

- Non-cryptographic hash function
- 128 bit hash
- Very low collision rate
- Fast

Deduplication SW

- Basic deduplication process in SW

```
Do {
  Read data from file
  Perform chunking (rollingHash)
  Calculate fingerprint (murmurHash)
  Save <hash, chunk> to DB
  Enqueue hash value into hash list
  Eliminate chunk data from buffer
} while (!file.eof());
Save <filename, hash list> pair to DB
```

Calculate murmurHash
→ 0xe045f78ac…

Hardware Design

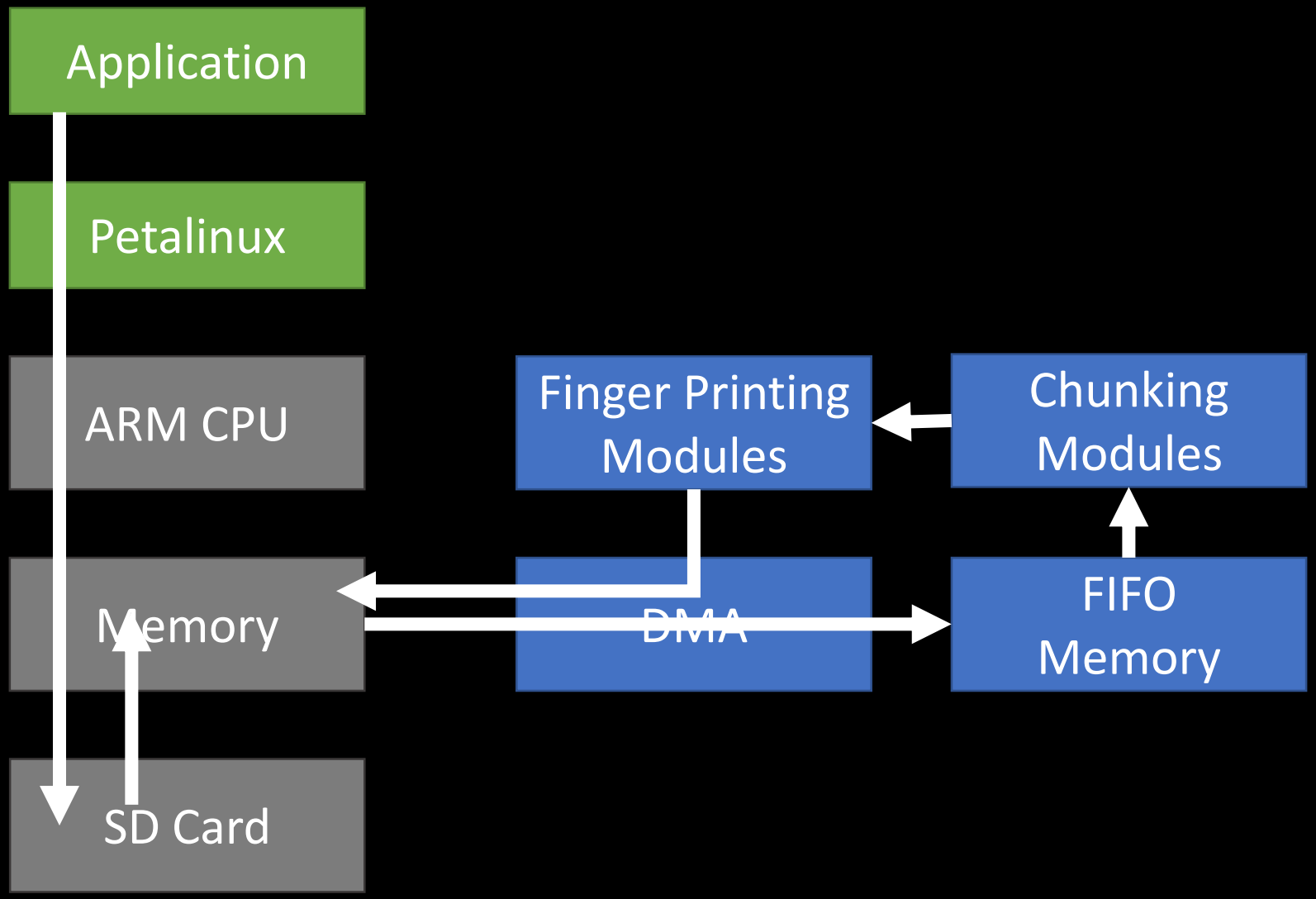
FPGA Key Challenges

- Performance disadvantage
  - Computation must be 7 times faster than CPU
    - PS 667Mhz VS PL 100Mhz
- Extra overhead when FPGA is used
  - Memory copy from PS to PL
  - Memory copy from PL to PS

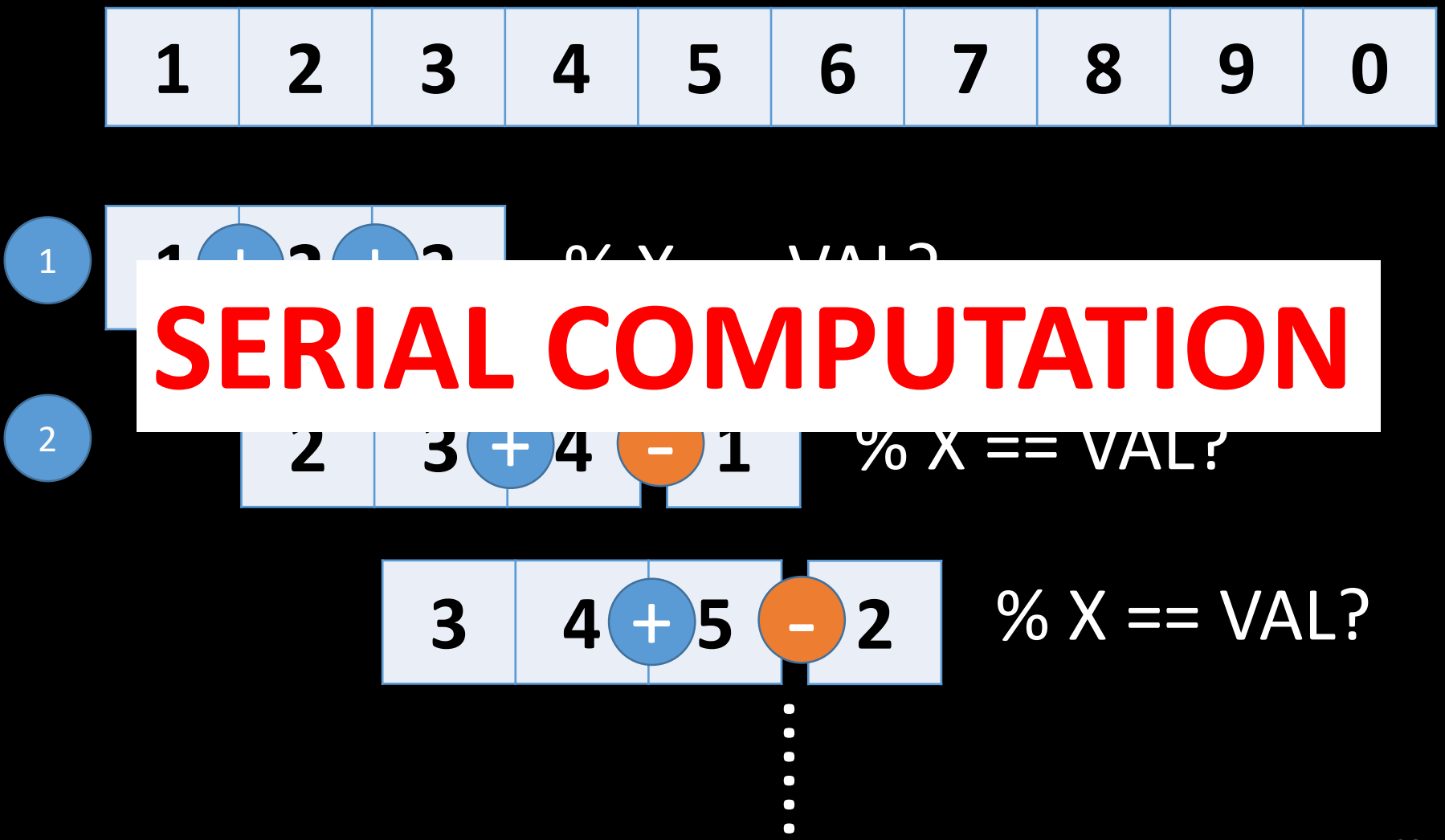
FPGA Key Technical Issues

- What to parallelize
  - Chunking hash
  - Finger printing hash
- How to parallelize
  - Unrolling
  - Memory placement
  - Pipelining
- Communication between PS and PL
  - DMA

Design in a Nutshell



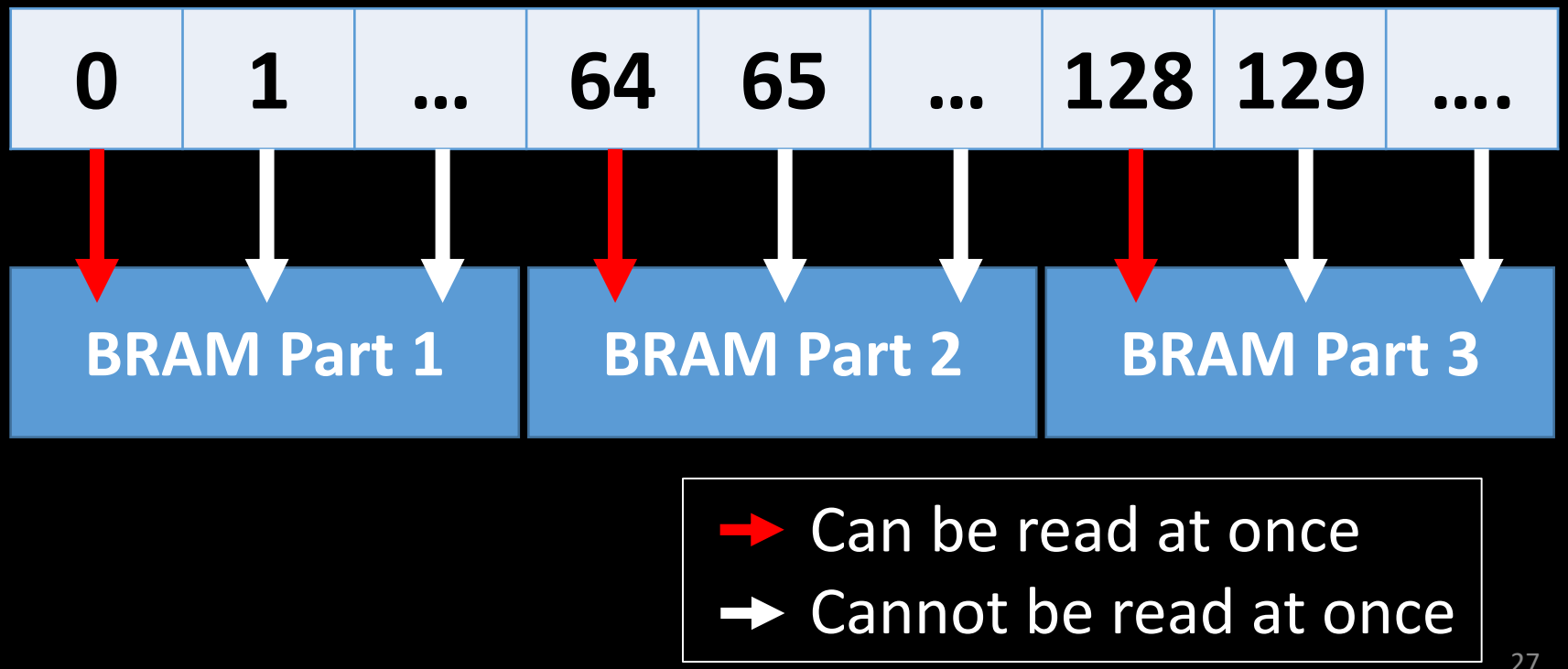
Recall Fast & Simple Rolling Hash



SERIAL COMPUTATION

BRAM Basics

- Maximum partition: 128
  - To parallelize, BRAM must be partitioned
    - Read per partitioned BRAM

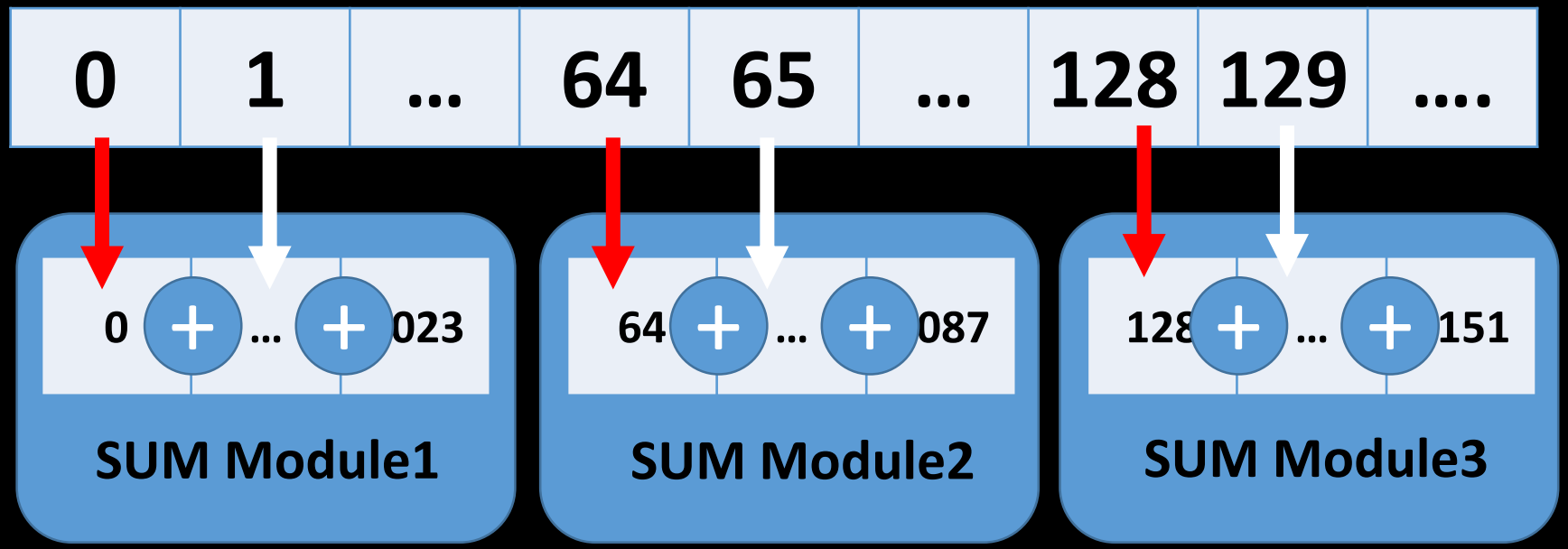


Chunking

Hardware Design

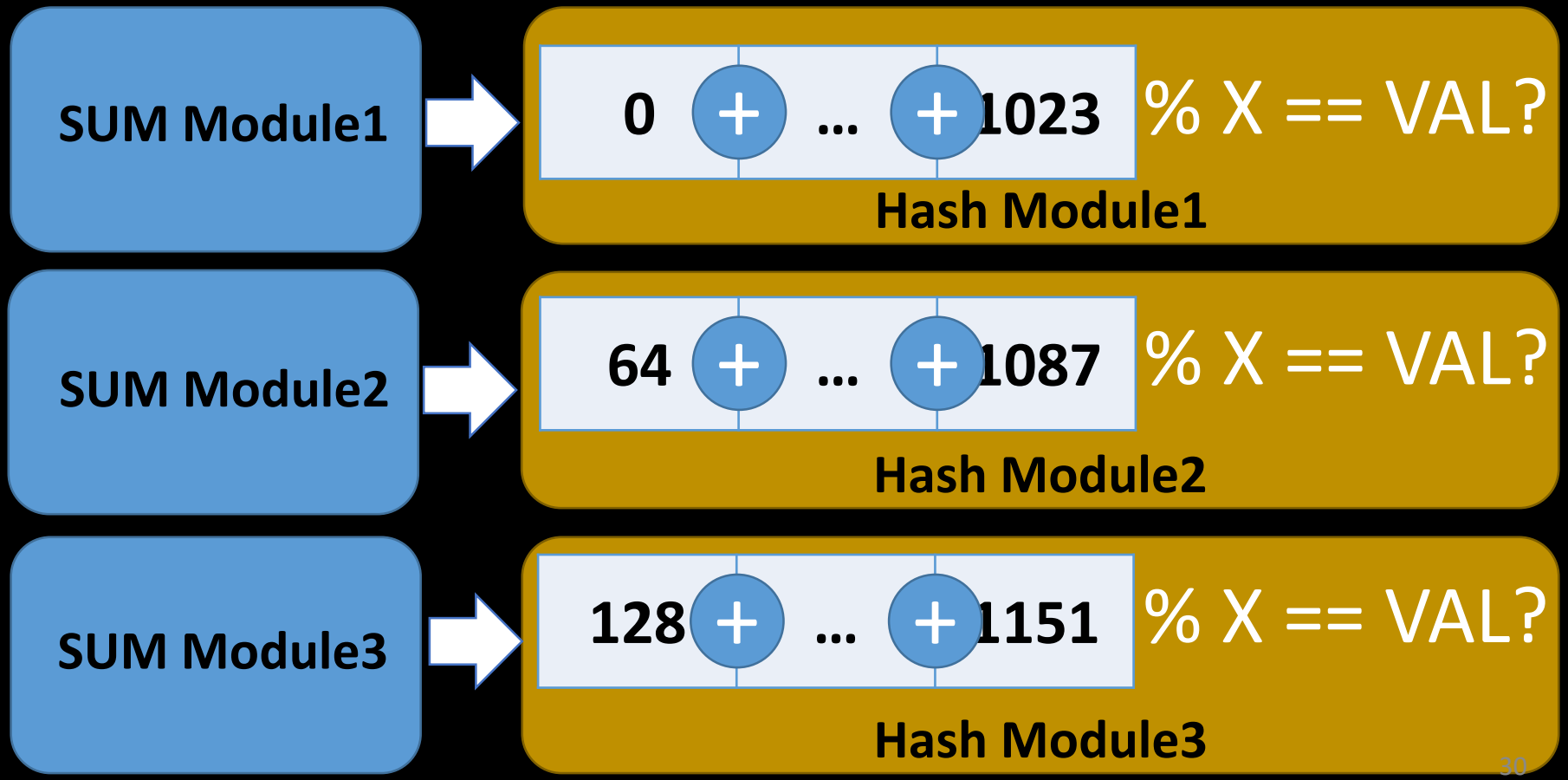
Parallelized Chunk Hash

- PS sends 8KB data to PL via DMA
- Partition 8KB data into multiple windows
- Unroll modules to calculate sum
  - Parallelized computation



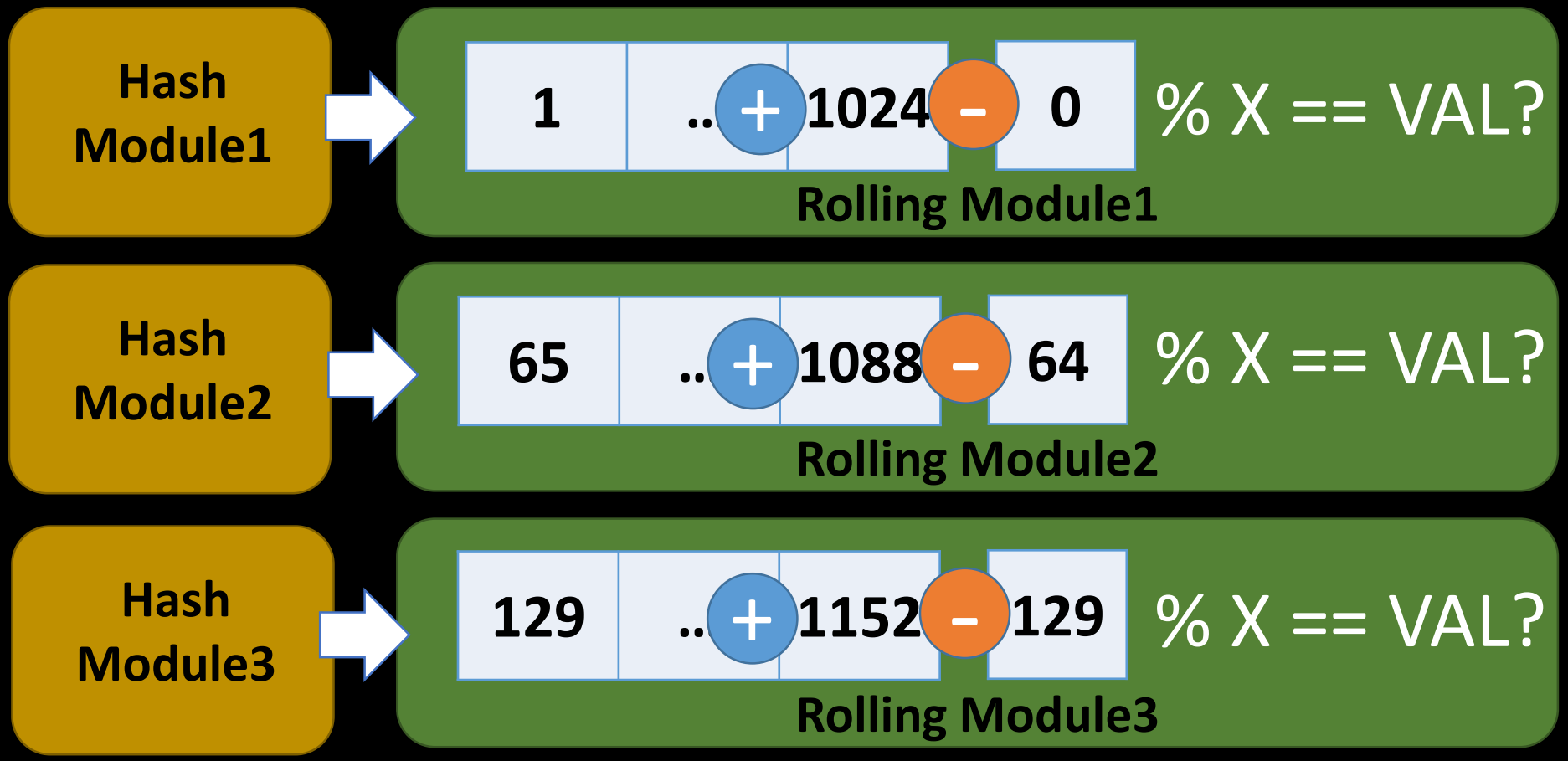
Parallelized Chunk Hash

- Compute hash of each added values
  - Parallelized computation



Parallelized Chunk Hash

- Compute rolling hash
  - Parallelized computation

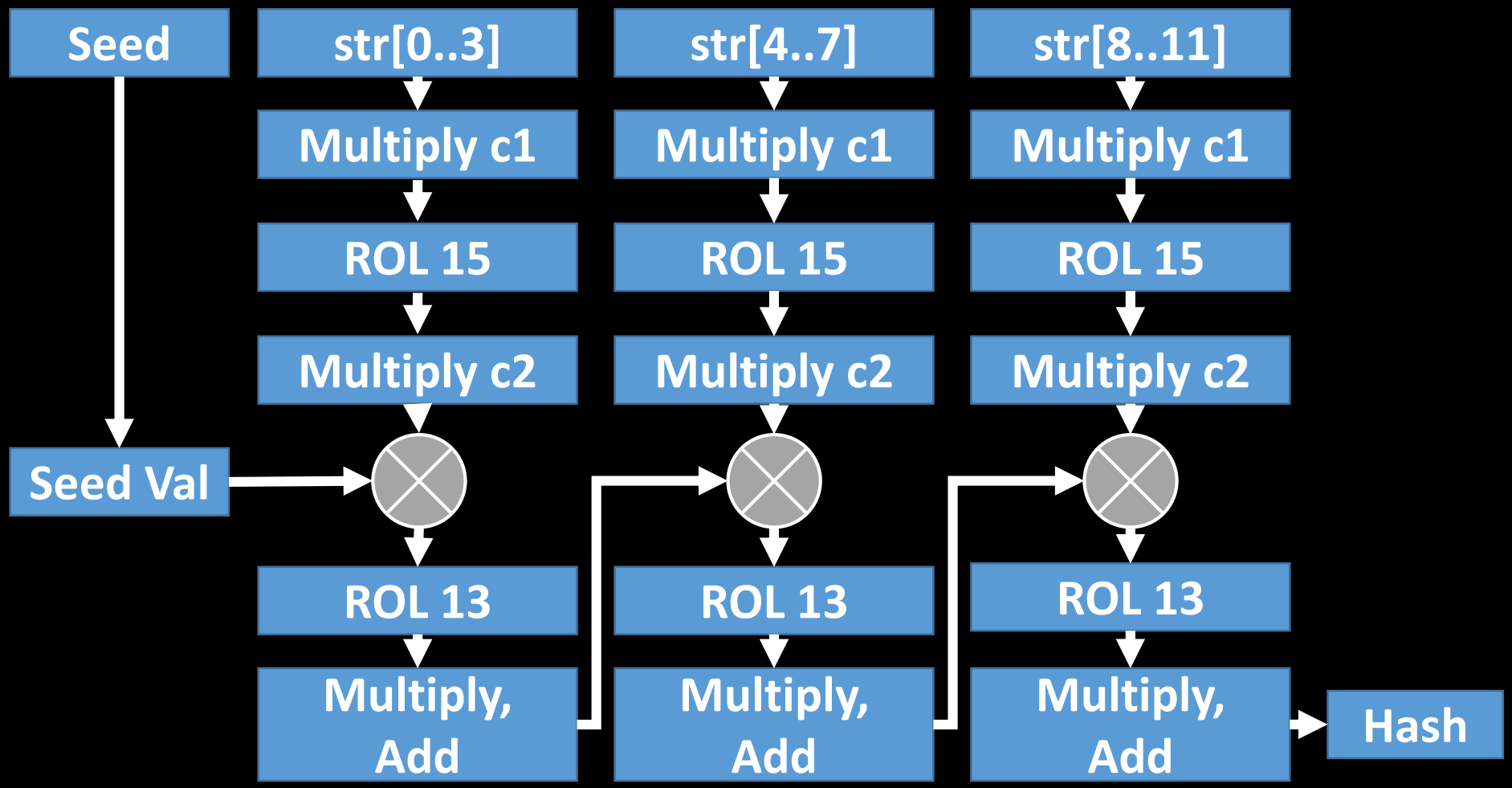


Finger Printing

Hardware Design

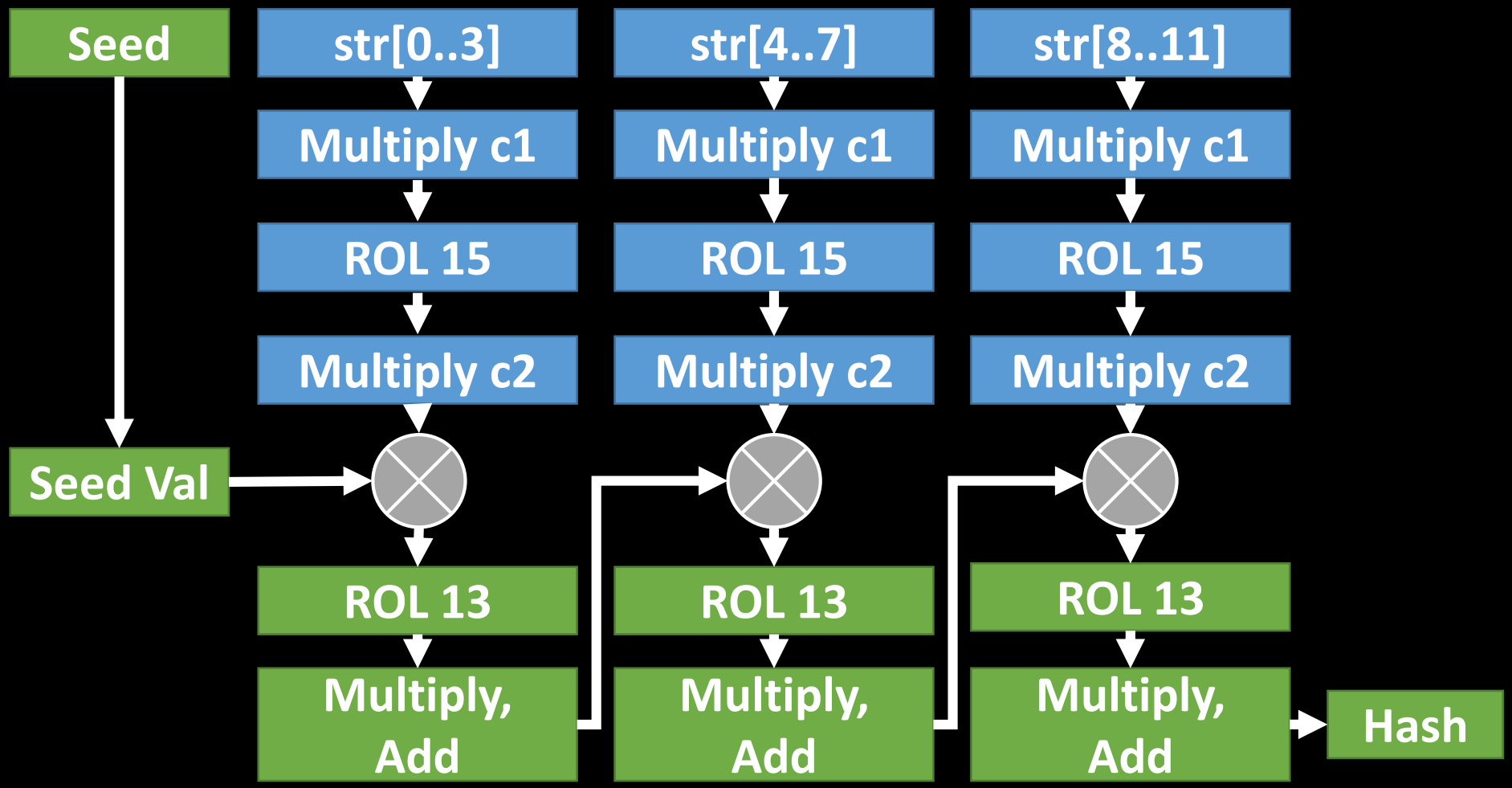
Parallelized Finger Printing

- Murmur hash for finger printing



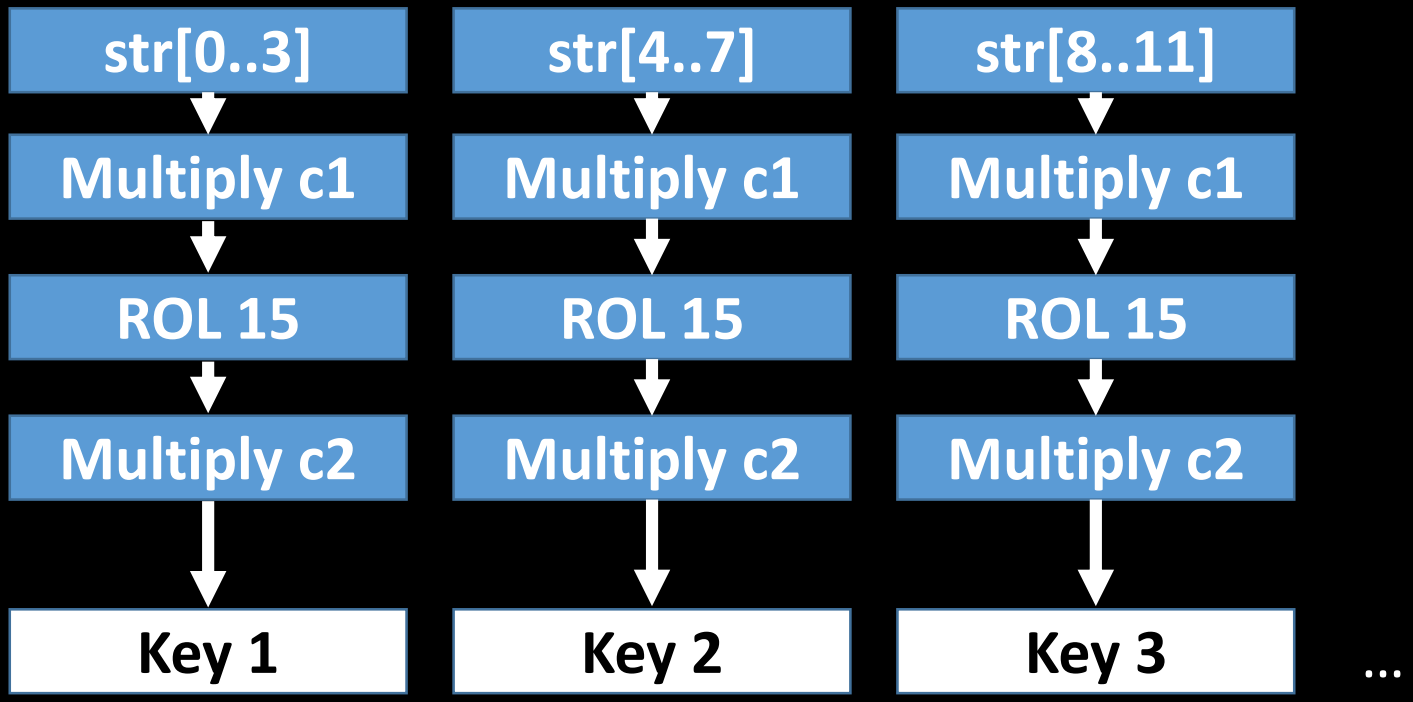
Parallelized Finger Printing

- Hash can partly be calculated in parallel



Parallelized Finger Printing

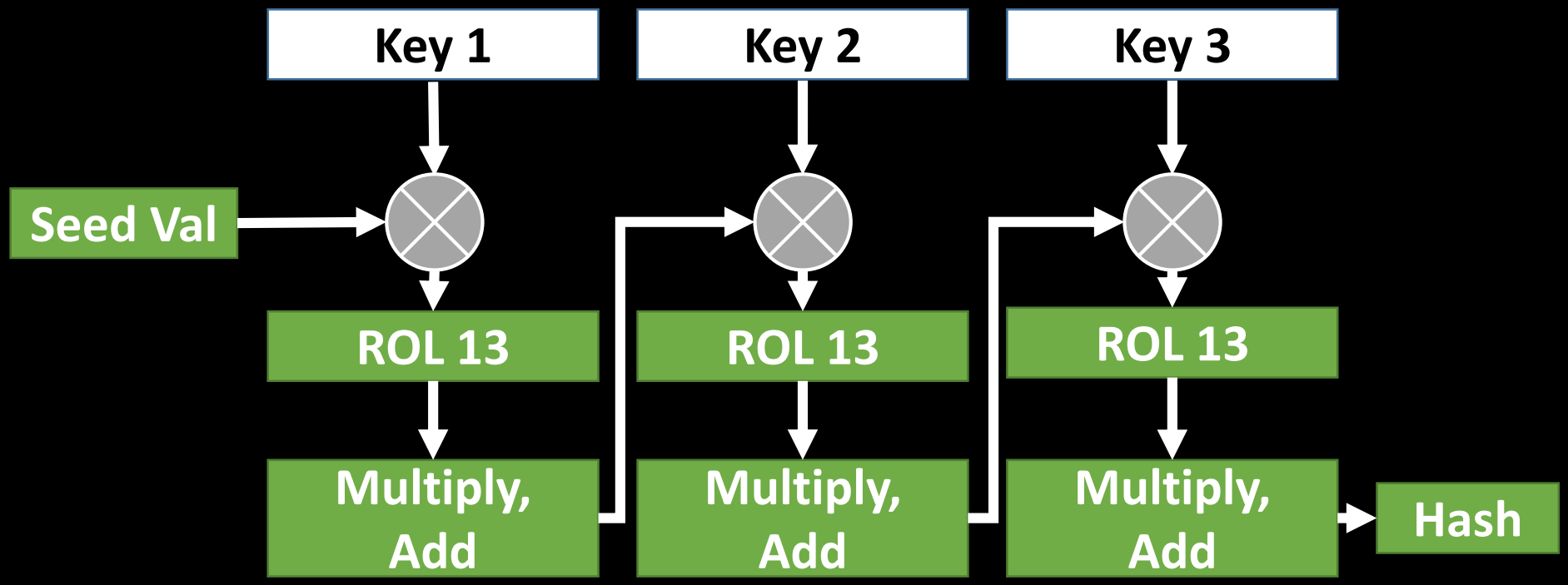
1. Compute parallelizable part simultaneously



Total up to 2048 keys (one key for 4 bytes data)
in 45 cycles

Parallelized Finger Printing

2. Serially compute non-parallelizable part



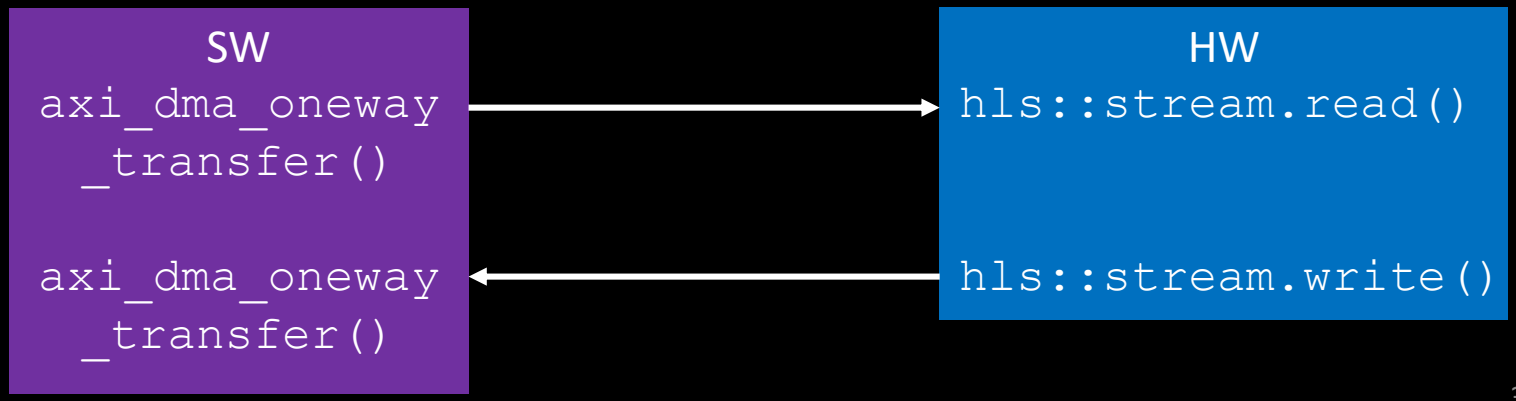
Calculate final hash
in 8194 cycles

DMA

Hardware Design

PS – PL DMA Communication

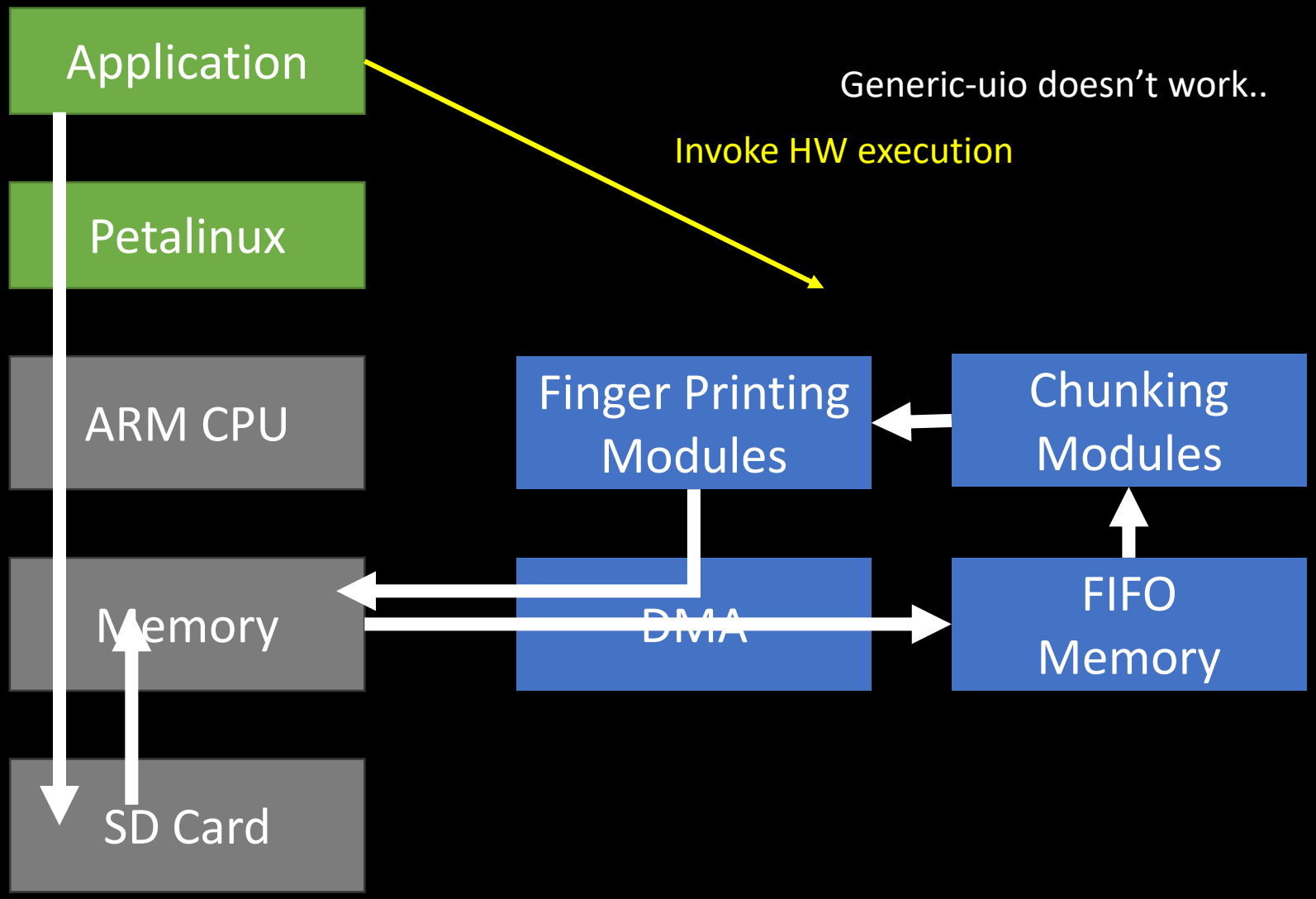
- Device driver for AXI DMA
  - Used AXI DMA library for Zynq in Github
  - https://github.com/bperez77/xilinx_axidma
  - `axi_dma_oneway_transfer(...)`
  - Actual transfer seems to be done when HW module calls `hls::stream.read()`



Device Driver

Hardware Design

Remaining Step



Device Driver for Custom HW

- Need a dedicated device driver for handling our HW module
- Why not use generic-uio device driver?
  - It seems not support custom interrupt handling
- Our device driver handles interrupt when `AP_DONE` signal becomes high

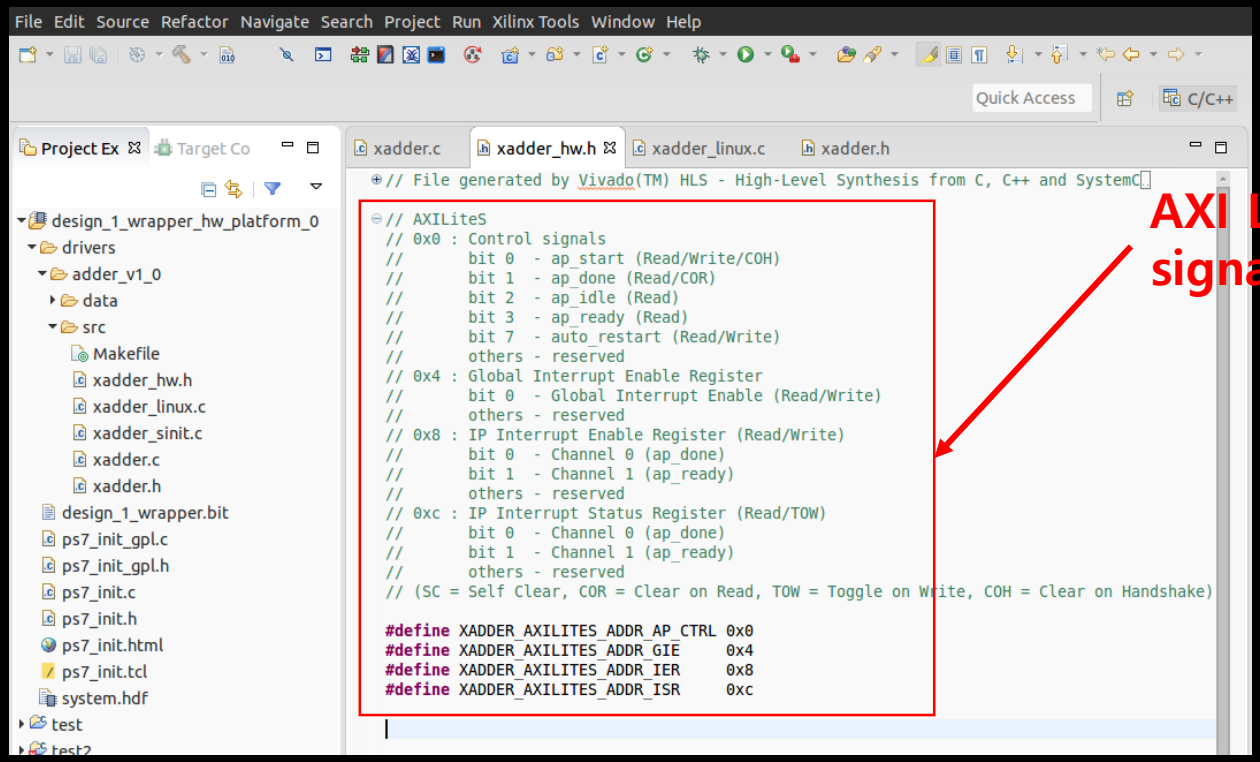
Registered interrupt for 'hello' HW module

```
163:        0          0        GIC  41 Edge      f8005000.watchdog
164:        1          0        GIC  63 Level     hello
165:        1          0        GIC  61 Level     xilinx-dma-controller
166:        1          0        GIC  62 Level     xilinx-dma-controller
```

Registered interrupt for AXI DMA

Device Driver for Custom HW

- HW module interface
  - For standalone, auto-generated by Vivado
  - Generated when you **export hardware**



Device Driver for Custom HW

- Example: How to invoke HW execution?
  - See device driver for standalone
  - `*(0x43c00000 + 0x0) |= 1;`

```
void XAdder_Start(XAdder *InstancePtr) {
    u32 Data;

    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

    Data = XAdder_ReadReg(InstancePtr->Axilites_BaseAddress, XADDER_AXILITES_ADDR_AP_CTRL) & 0x80;
    XAdder_WriteReg(InstancePtr->Axilites_BaseAddress, XADDER_AXILITES_ADDR_AP_CTRL, Data | 0x01);
}
```

- In Linux device driver, it is equal to

```
u32 reg = ioread32(0xe09a0000 + 0x0);
iowrite32(reg|0x1, 0xe09a0000);
```

```
hello 43c00000.adder: Device Tree Probing
hello 43c00000.adder: hello at 0x43c00000 mapped to 0xe09a0000
root@petalinux_customip:~#
```

Merge Deduplication HW/SW

- Basic deduplication process in SW/HW

```
Do {
  Read data from file to fill 8KB buffer
  Transfer buffer data via DMA to PL
  Perform chunking to get a chunk
  Invoke PL execution and wait to be completed
  Calculate SHA1 hash for the chunk
  Transfer result hash via DMA from PL
  Save <hash, chunk> pair to DB
  Enqueue hash value into the hash list
  Eliminate chunk data from the buffer
} while (!file.eof());
Save <filename, hash list> pair to DB
```

- Chunking & Hashing are accelerated by HW

Result

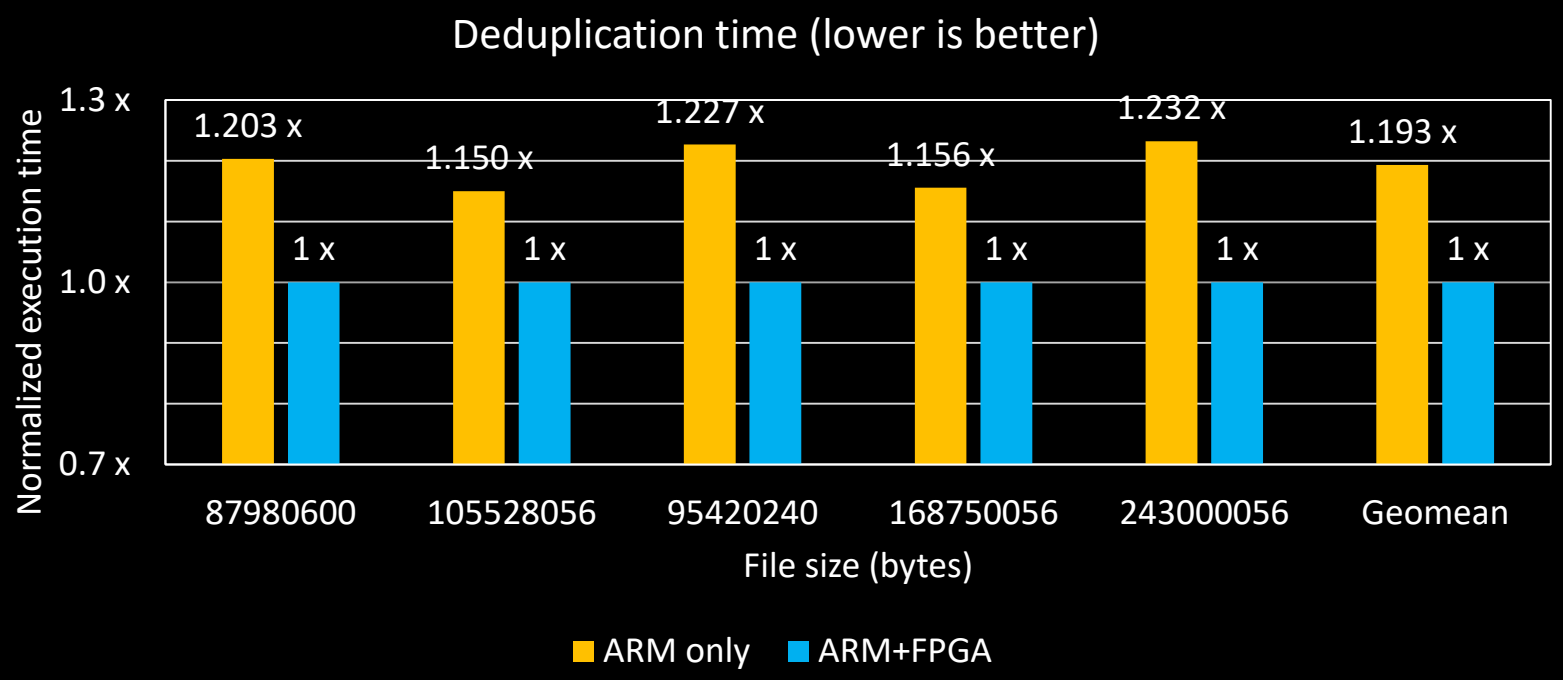
Experimentation

- Compare 3 environment
  - Intel x86_64
    - i7-6700
  - ARM only system (SW based Deduplication)
    - ARM cortex A9
  - ARM + FPGA (HW based Deduplication)

	Intel x86_64	ARM cortex PS	ARM + FPGA
Clock frequency	3.4GHz	667MHz	100MHz
Ratio (slower)	1	5.14	34.82

- Workload : 5 image files
  - 88MB ~ 243MB

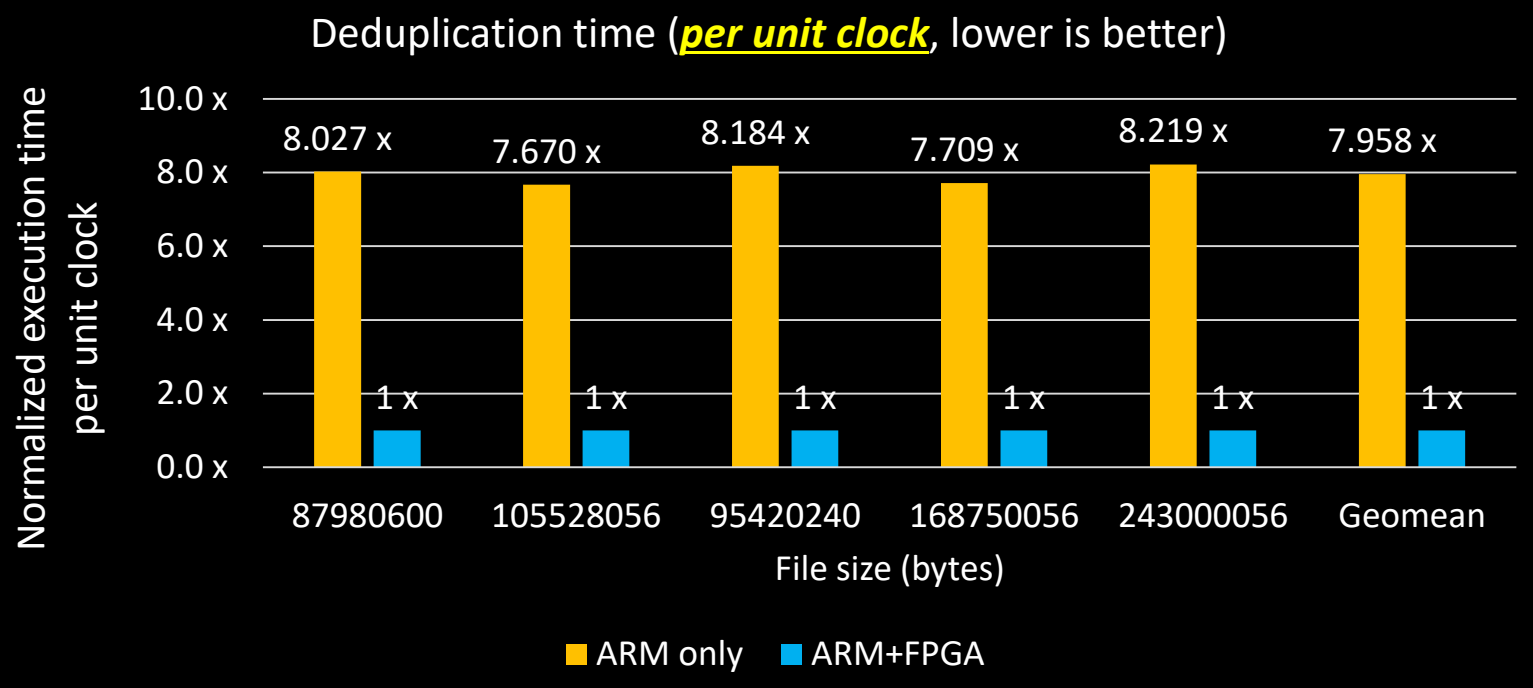
Performance : ARM vs PL [1/2]



	ARM cortex PS	ARM + FPGA
Clock freq. ratio	1 (667MHz)	6.6x (100MHz)

- Even Clock freq. 6 x slower than ARM SW only
- ARM + FPGA shows 1.2x better performance

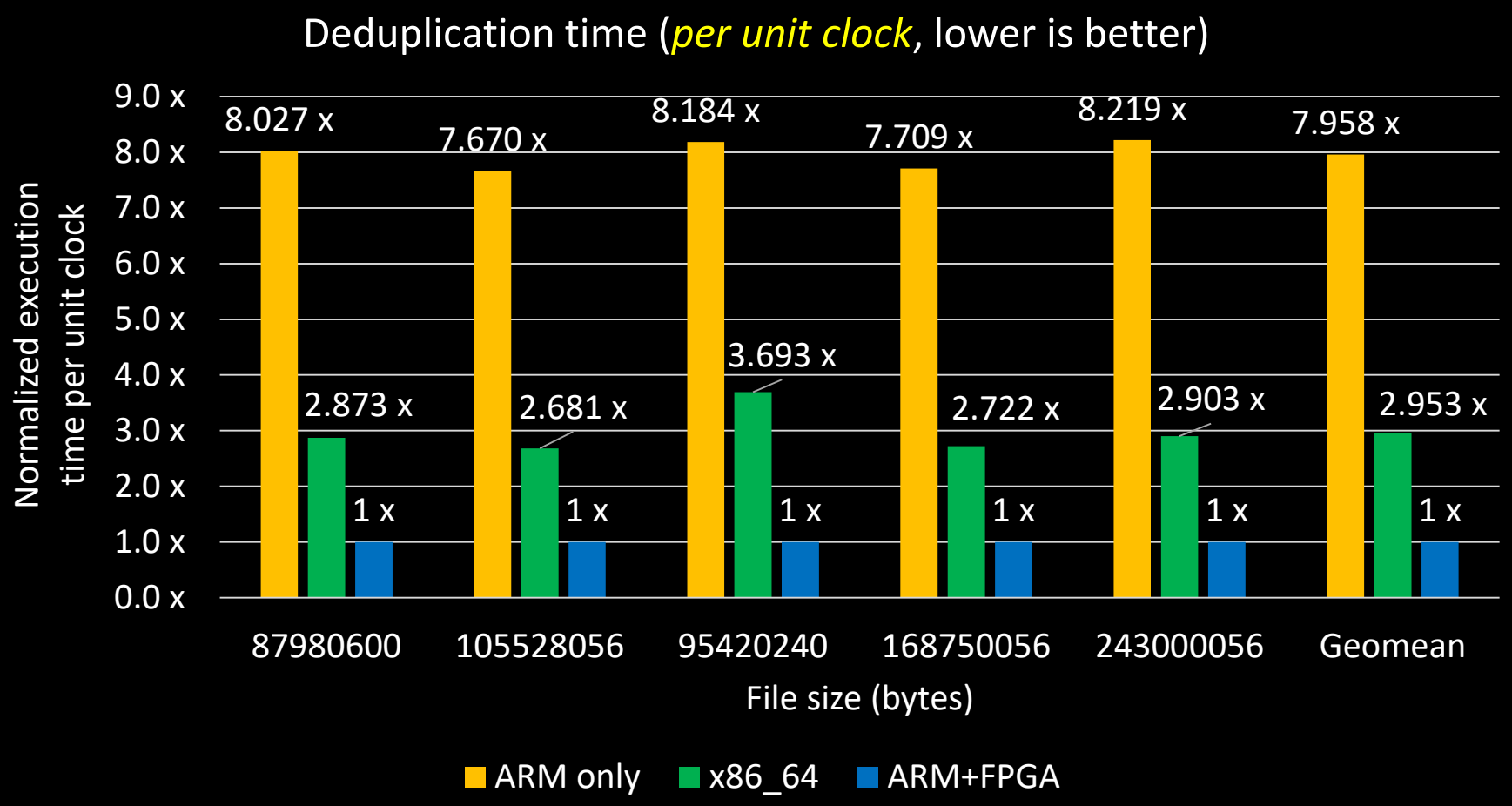
Performance : ARM vs PL [2/2]



	ARM cortex PS	ARM + FPGA
Clock freq. ratio	1 (667MHz)	6.6x (100MHz)

- 8x better performance per unit clock

Performance : Overview

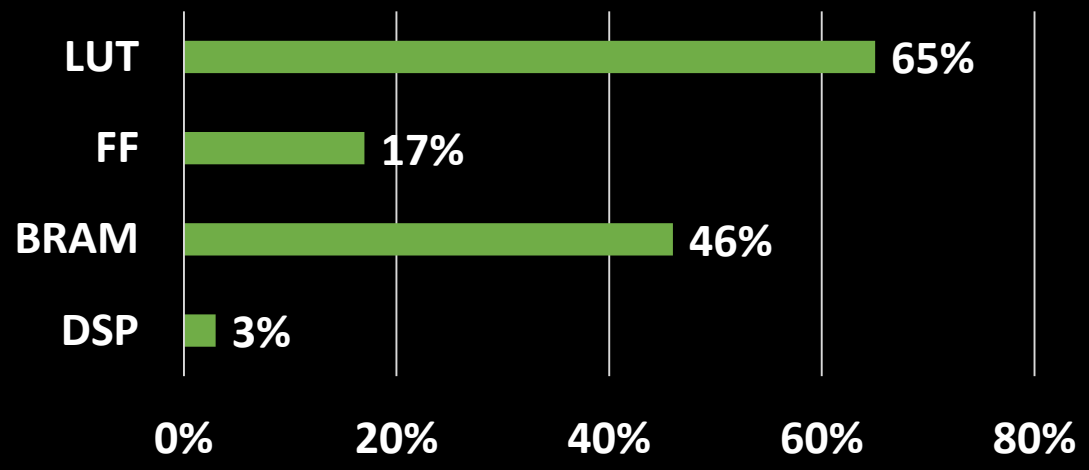


IP Utilization & latency

- Latency (clock cycle)

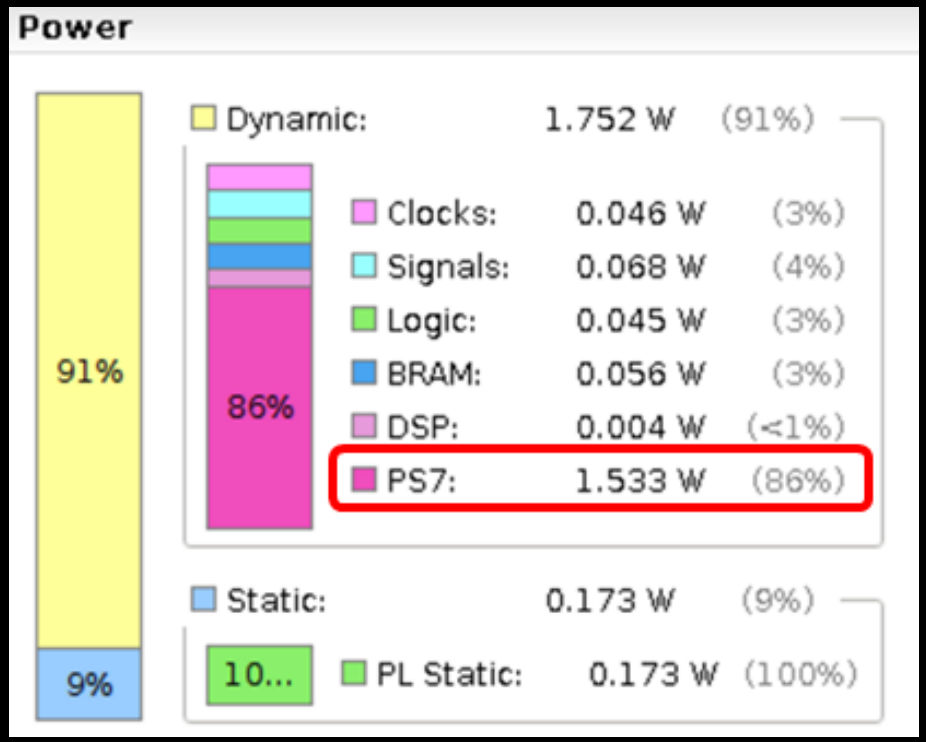
Latency		Interval		
min	max	min	max	Type
25212	25212	25213	25213	none

- Estimated utilization



Power Consumption

- Total on-chip power : 1.925W
  - Dynamic : 1.752W
  - Static : 0.173W



Executive Summary

- ✓ A-Z development of deduplication
  - ✓ SW version of deduplication
  - ✓ Chunking HW logic
  - ✓ Fingerprinting HW logic
  - ✓ HW device driver on Petalinux
  - ✓ Merge deduplication SW-HW

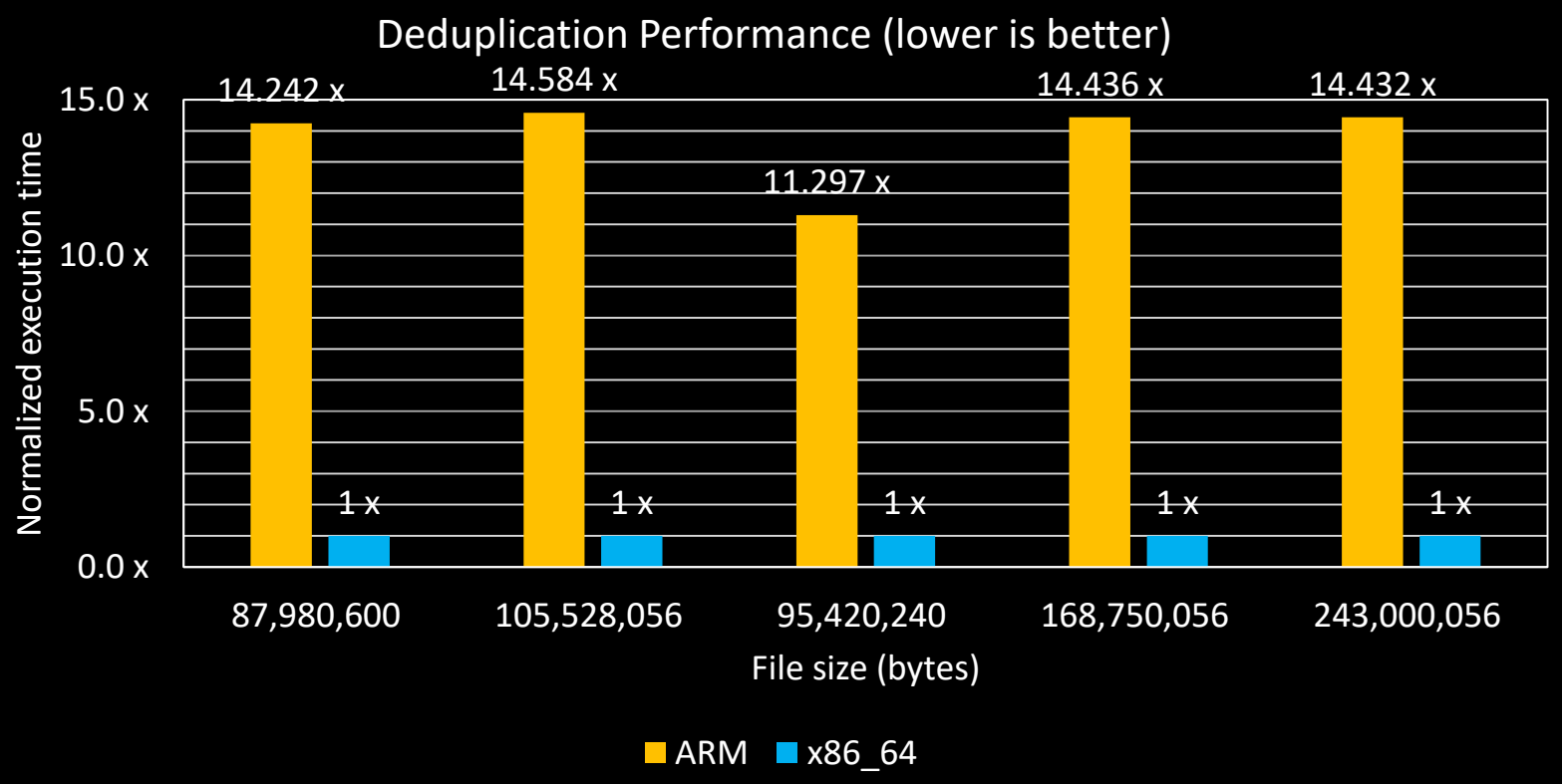
- ✓ Performance Benefit
  - ✓ ARM+FPGA
    - ✓ 8x faster than ARM only
    - ✓ 3x faster than x86_64
  - ✓ With low power consumption

Thank you

Backup

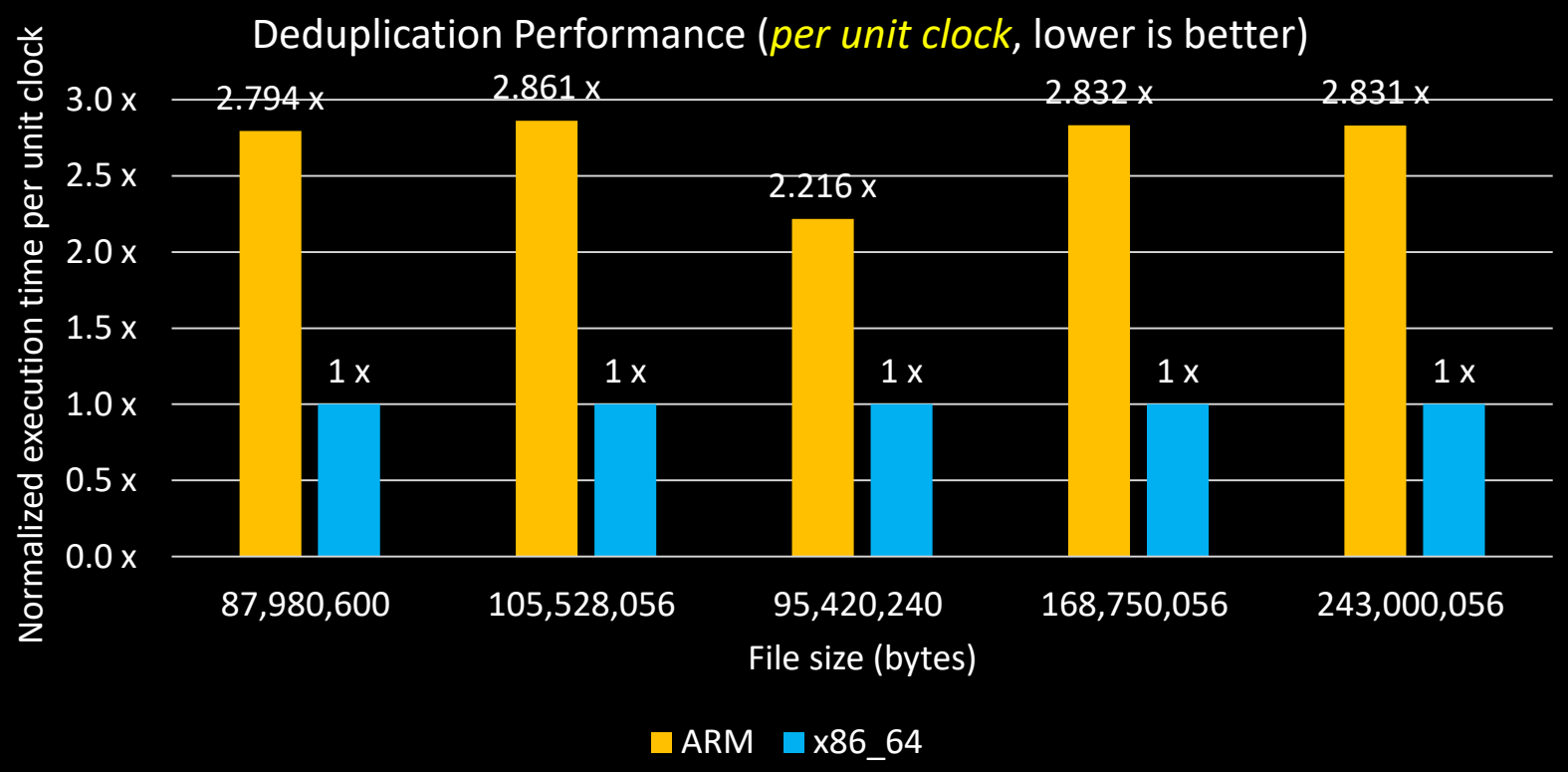
Performance : ARM vs x86_64 [1/2]

	ARM cortex PS	Intel x86_64
Clock frequency	667MHz	3.4GHz



Performance : ARM vs x86_64 [2/2]

	ARM cortex PS	Intel x86_64
Clock frequency	667MHz	3.4GHz



Implemented Design