FLETCHER

Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow

@ FPL2019, Barcelona, September 11, 2019

Johan Peltenburg[1], Jeroen van Straten[1], Lars Wijtemans[1]
Lars T.J. van Leeuwen[1], Zaid Al-Ars[1], H. Peter Hofstee[2]

1. Delft University of Technology, Netherlands
2. IBM, Austin, Texas, USA

Thanks to our supporters:

Fitoptivis European ECSEL project
no. ECSEL2017-1-737451

Xilinx

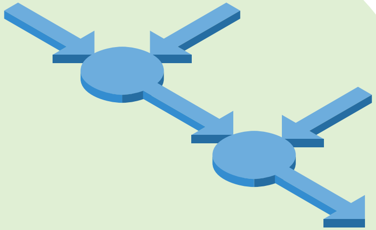
FIT PTI VIS

TU Delft

Outline

- The challenge of FPGA integration with Big Data Analytics
- Overcoming serialization bottlenecks with Apache Arrow
- Fletcher
- Mini-tutorial (if time)
- Results
- Conclusion & future work

An FPGA Accelerator Dev. Perspective

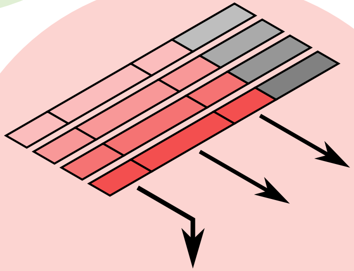


High-performance datapath

Write a host-side C lib

01001011

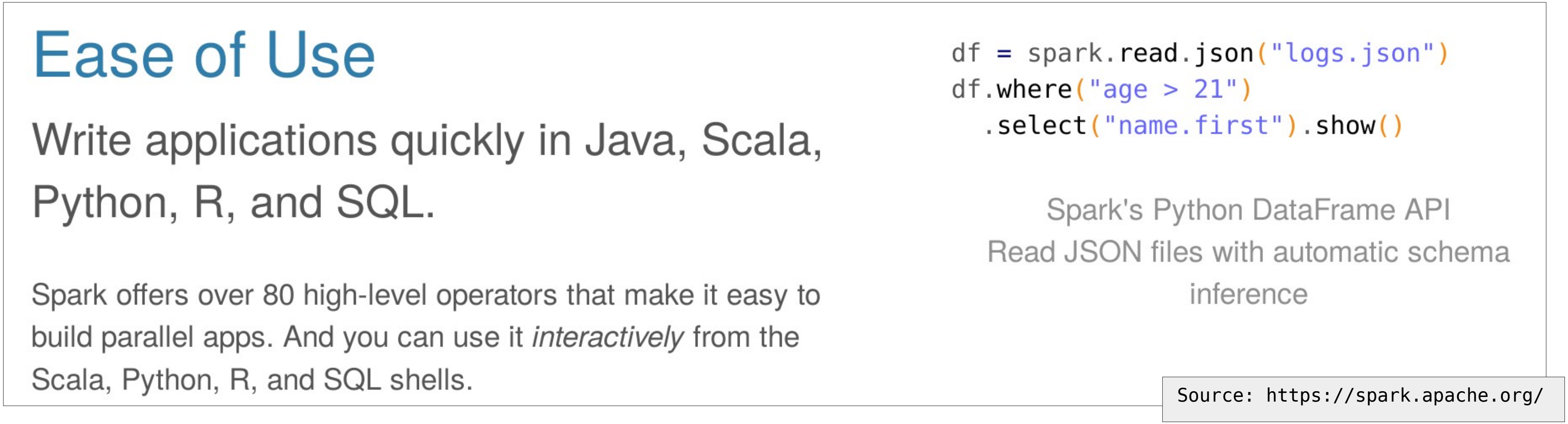
Byte or even bit-level control of data structure In memory



Structs, unions, bitspecs, padding

Sculpt datastructure to feed datapath efficiently

A Big Data Analytics Dev. Perspective:



Ease of Use

Write applications quickly in Java, Scala, Python, R, and SQL.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala, Python, R, and SQL shells.

```
df = spark.read.json("logs.json")
df.where("age > 21")
  .select("name.first").show()
```

Spark's Python DataFrame API
Read JSON files with automatic schema inference

Source: https://spark.apache.org/

- DataFrame: like a database table or excel spreadsheet, but...

- Huge. Typically **in the order of GiBs to TiBs**.
- **Distributed** over multiple worker nodes (also in storage).
- Operations on it build Directed Acyclic Graphs (DAGS) and are lazily evaluated.
- DAGs are **optimized, planned and scheduled to exectue in parallel over a cluster**.
- **Resilient** to node failure, provides automatic recovery and continuation.

- What is all that computer scientist magic that makes this possible?

Big Data Analytics SW Ecosystem

Frameworks for
storage, scalability,
resilience, analysis,
etc..

APACHE Spark™

hadoop

DASK

pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

dremio

Parquet

Software
languages
& run-times

R

python™

Java™

Scala

MATLAB

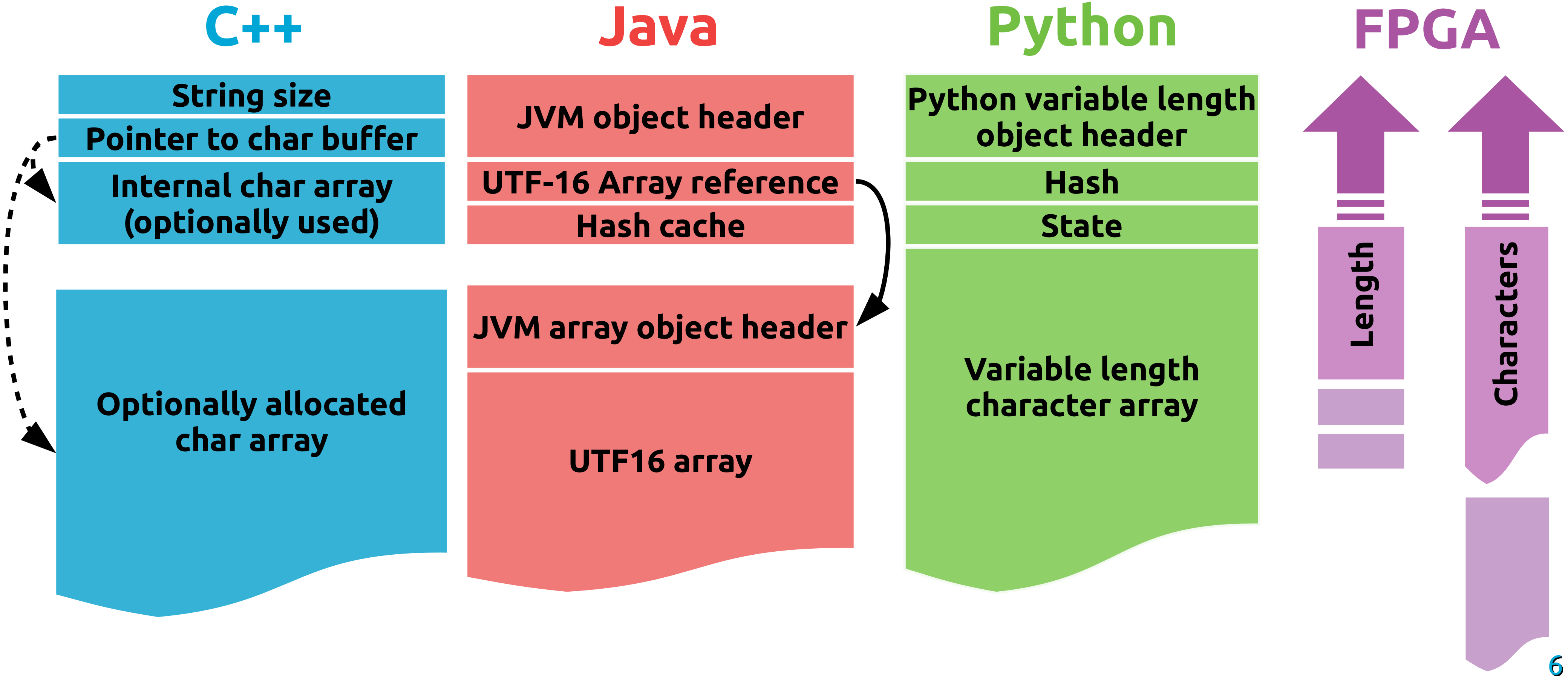
C#

Rust

GOLANG

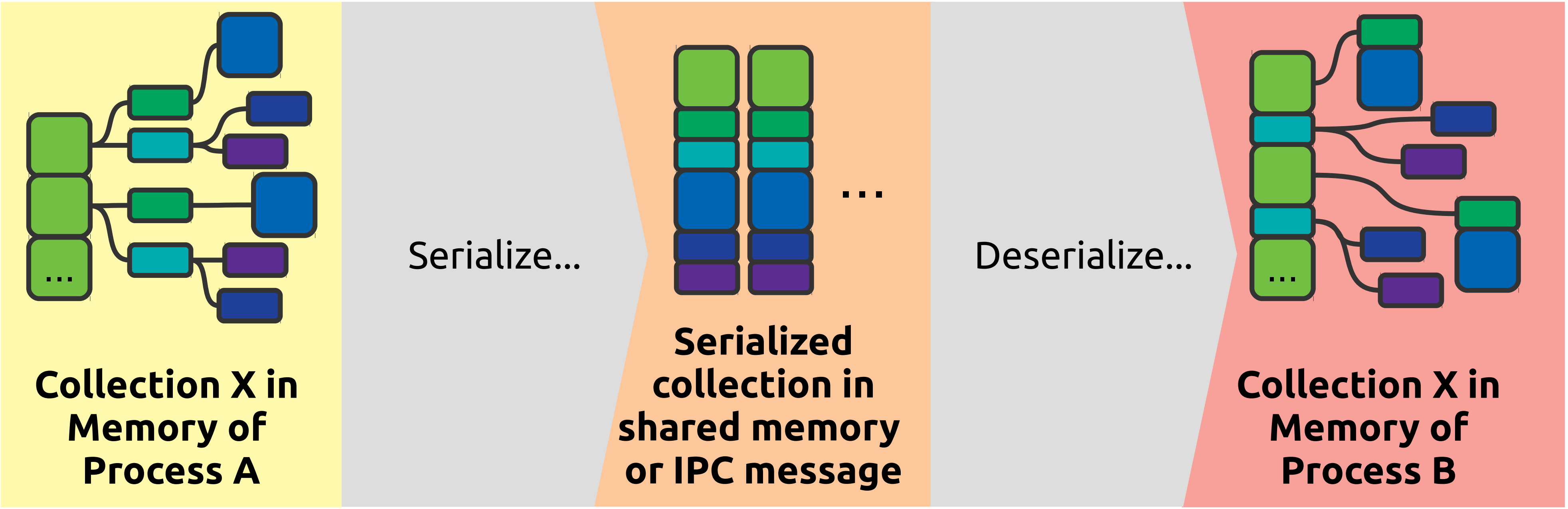
C++

A string



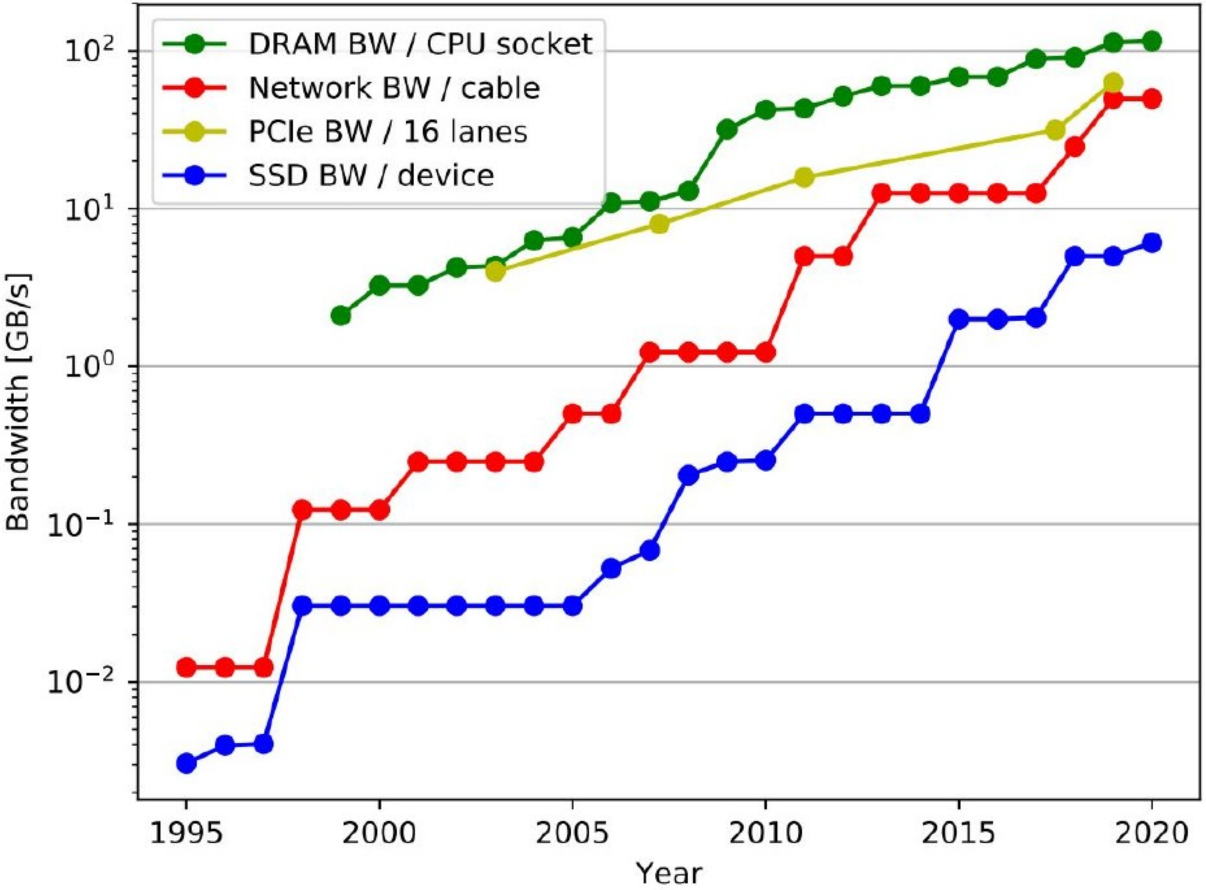
C++
String size
Pointer to char buffer
Internal char array (optionally used)
Optionally allocated char array
Java
JVM object header
UTF-16 Array reference
Hash cache
JVM array object header
UTF16 array
Python
Python variable length object header
Hash
State
Variable length character array
FPGA
Length
Characters

Serialization


...
Collection X in Memory of Process A
Serialize...
...
Serialized collection in shared memory or IPC message
Deserialize...
...
Collection X in Memory of Process B

- Iterate over all objects in collection (data is big)
- Traverse all object graphs (memory latency)
- Copy fields to some intermediate format both A and B understand (bandwidth lost)
- Reconstruct objects in B ((de)allocation overhead)

I/O bandwidth catching up

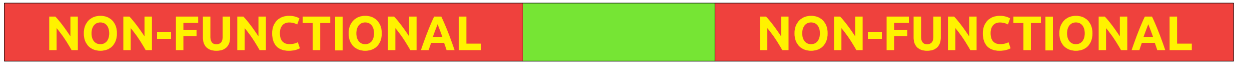


[1] F. Kruger, “CPU Bandwidth The Worrisome 2020 Trend,” Mar. 2016. [Online]. Available: https://blog.westerndial.com/cpu-bandwidth-the-worrisome-2020-trend/

Relative impact on accelerators

Original process on CPU:

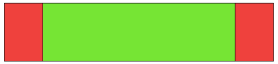
Process on GPGPU/FPGA with serialization (potentially, but not necessarily, exaggerated)



NON-FUNCTIONAL

NON-FUNCTIONAL

Desired profile:

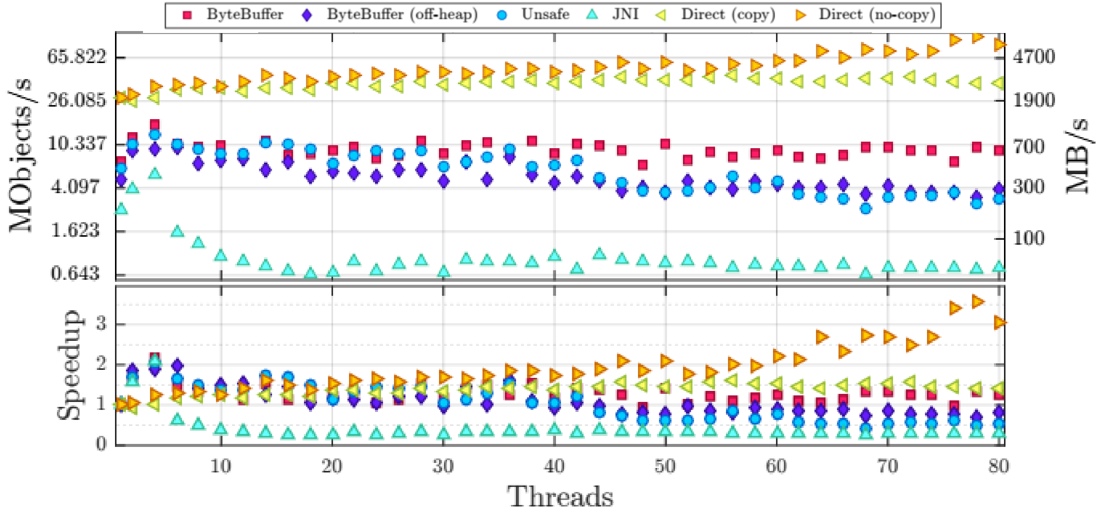


🟦 CPU compute time

🟥 (De)serialize / copy time

🟩 Accelerator compute time

Serialization throughput on collection of Java (OpenJDK) objects on POWER8 [1]:

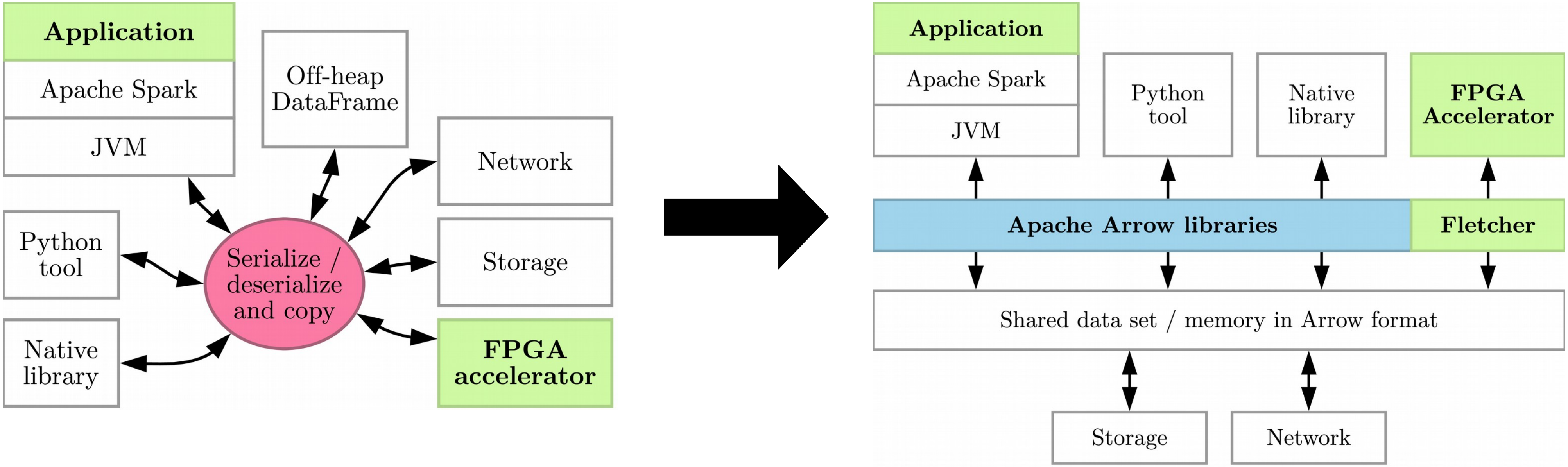


[2] J. Peltenburg, A. Hesam, and Z. Al-Ars, “**Pushing Big Data into Accelerators: Can the JVM Saturate Our Hardware?**” in International Conference on High Performance Computing. Springer, 2017, pp. 220–236.

Overcoming serialization bottlenecks

- In-memory formats determined by:
  - Programming languages
    - Run-time system design choices
    - Standard libraries
  - Algorithms
  - Programmers
- Increased heterogeneity → more IPC → more serialization overhead
- What if data is…
  - In a standardized format?
    - That every language can use (through libraries or otherwise).
  - As contiguous as possible?
    - We can move it using large bursts, no pointer chasing, less misalignment overhead

Apache Arrow[3]



- Standardized representation in-memory – Common Data Layer
- Columnar format
  - Hardware friendly while iterating over entries in single column (SIMD, caches, etc…)
  - Better for many algorithms, worse for some others.
- Libraries and APIs for 10+ languages to build and access data sets (zero-copy)

[3] The Apache Software Foundation, “Apache Arrow,” 2018. [Online]. Available: https://arrow.apache.org

Arrow in-memory dataset, trivial example

```
Schema MySchema {
   A: Float (nullable)
   B: List<Char>
   C: Struct{
         E: Int16
         F: Double
      }
}
```

Arrow terminology:

Schema:

Description of data types in a ***RecordBatch***

RecordBatch:

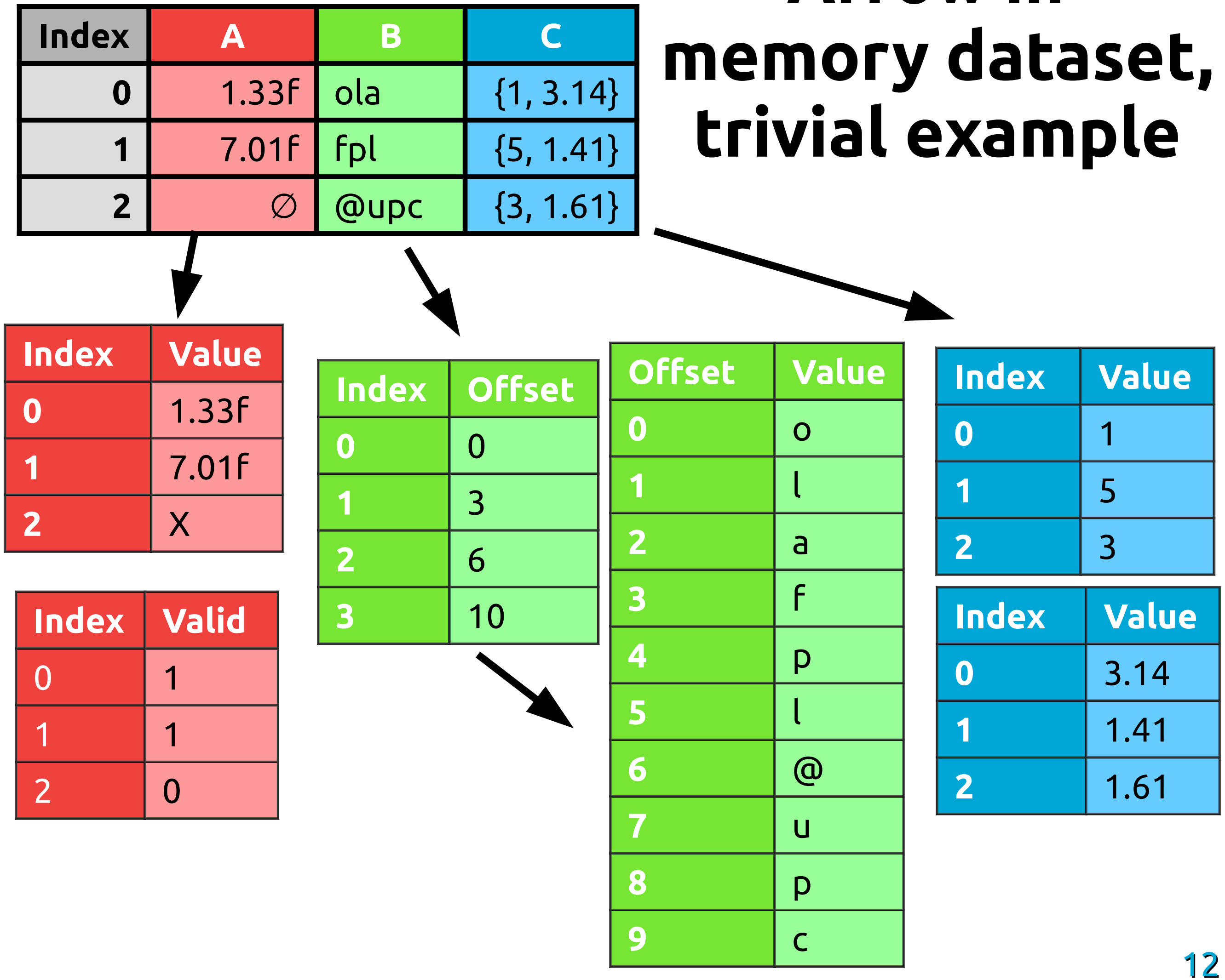
Tabular structure containing Arrow ***arrays***

Arrays:

A RecordBatch “column”. Combination of Arrow ***buffers***, can be nested

Buffers:

Contiguous C-like arrays



Index	A	B	C
0	1.33f	ola	{1, 3.14}
1	7.01f	fpl	{5, 1.41}
2	∅	@upc	{3, 1.61}

Index	Value
0	1.33f
1	7.01f
2	X

Index	Valid
0	1
1	1
2	0

Index	Offset
0	0
1	3
2	6
3	10

Offset	Value
0	o
1	l
2	a
3	f
4	p
5	l
6	@
7	u
8	p
9	c

Index	Value
0	1
1	5
2	3

Index	Value
0	3.14
1	1.41
2	1.61

Integrating FPGA and Arrow

- Arrow ‘turns out’ to be hardware-friendly
  - In-memory format clearly specified, to every bit
  - Highly contiguous & columnar format
    - Iterate over a column in streaming fashion
    - Useful for: maps, reductions, filters, etc...
  - Parallel accessible format
    - E.g. uses offsets, not lengths, for variable length data – we can start anywhere
    - Useful for: maps, reductions, filters, etc…
- Backed by a large and ever growing community
- Integration in many BDA frameworks, even without official format stability
- Can we generate easy-to-use, high throughput hardware interfaces automatically?

APACHE Spark

DASK

pandas

Parquet

...

Main contribution:

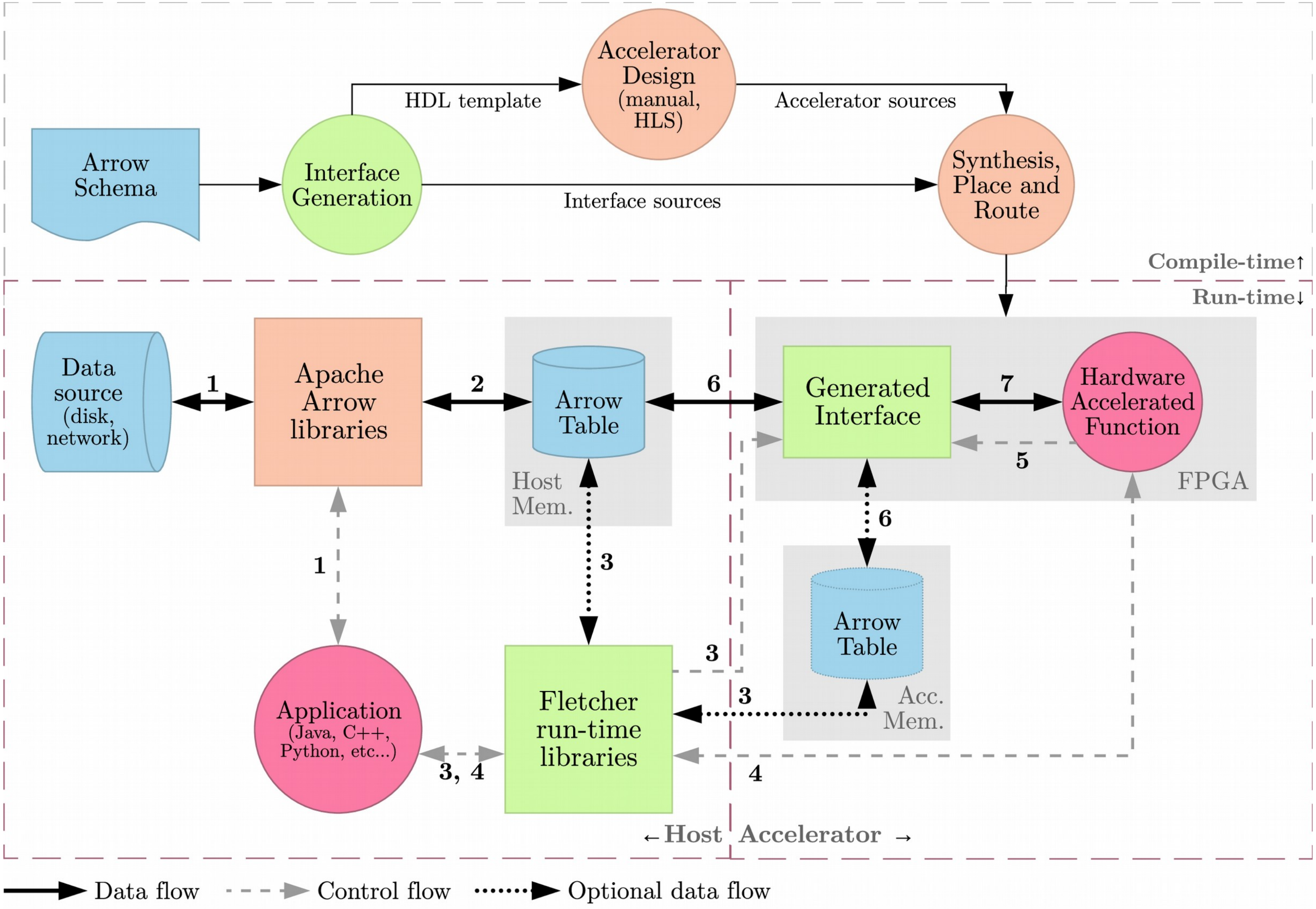
FLETCHER

Fully open-source (Apache-2.0),

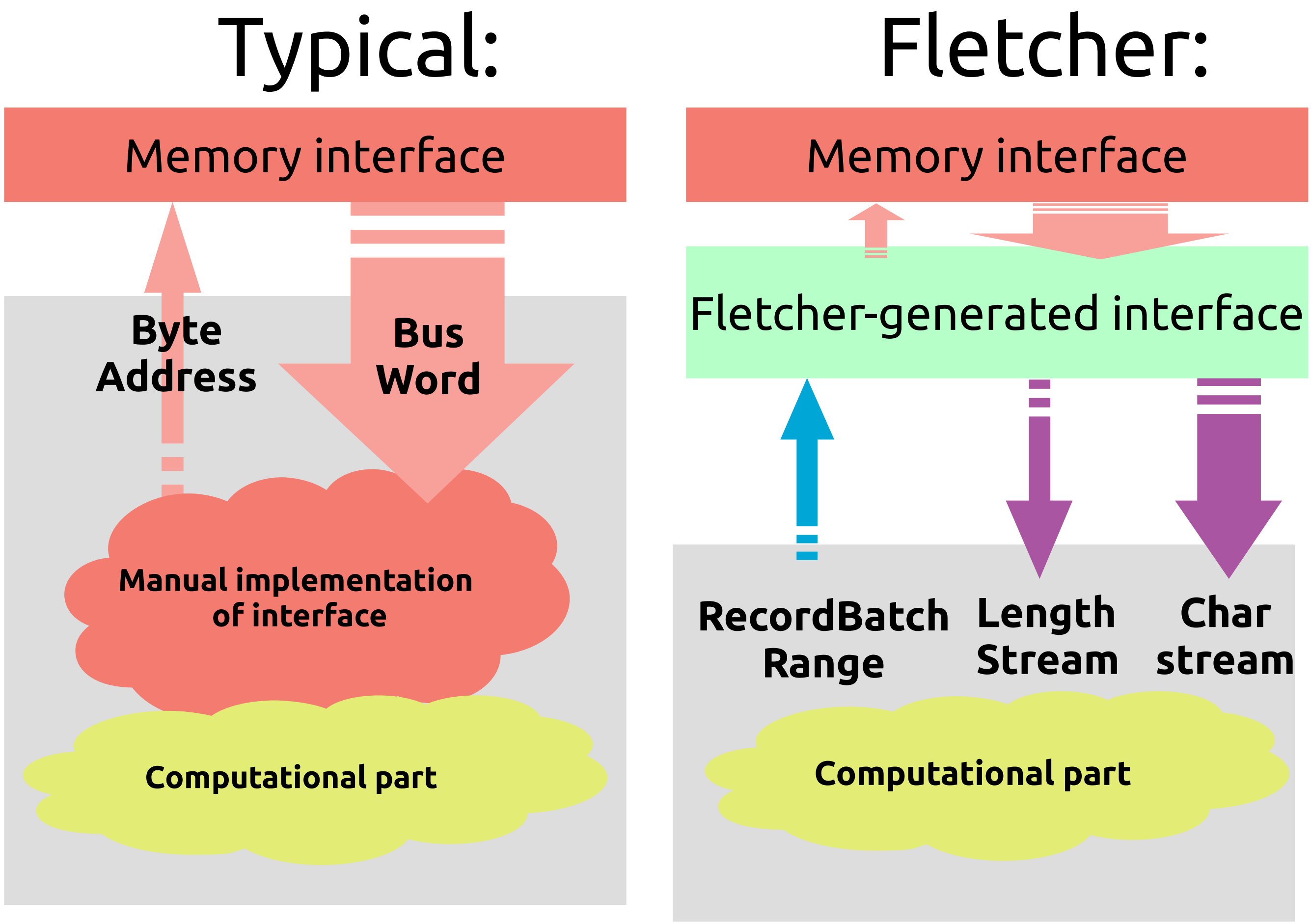
Vendor agnostic,

Generates easy-to-use high-throughput Interfaces.

Integrate FPGA accelerators with Apache Arrow.



Example: Interface for accelerator parsing strings



Easy-to use:
Data is delivered as streams that make sense w.r.t. schema field types.

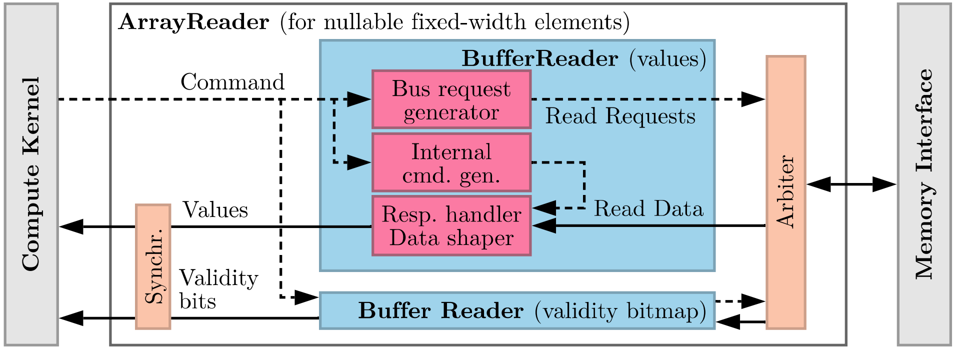
High-throughput:
Number of values delivered per cycle configurable

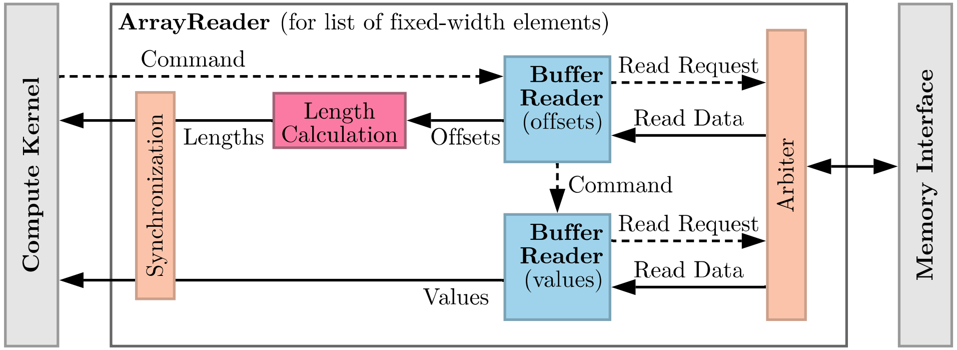
Generated interface overview

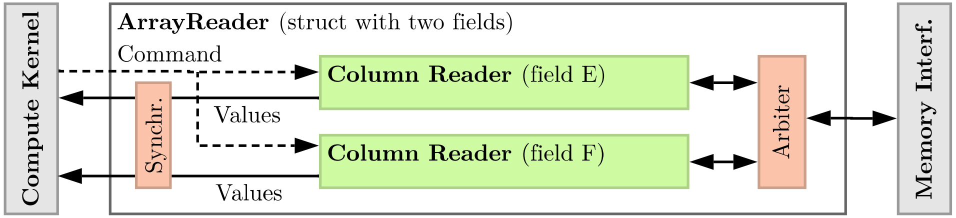
- Architecture based on library with streaming primitives
- BufferReader/Writer : Basic unit to read (N) Arrow Buffer elements
- ArrayReader/Writer : Combination of BufferReaders/Writers [1]
  - Dictated by the schema field and format specification
  - Generated through pure HDL; vendor agnostic
- RecordBatchReader/Writer : Combination of ArrayReaders/Writers
- Mantle : Wraps multiple RecordBatchR/W + bus infrastructure

[4] J. Peltenburg, J. van Straten, M. Brobbel, H. P. Hofstee, and Z. Al-Ars, “**Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow**”, in Applied Reconfigurable Computing, Cham: Springer International Publishing, 2019, pp. 32–47.

Combining BufferReaders into ArrayReaders







Index	Value
0	1.33f
1	7.01f
2	∅

Index	Valid
0	1
1	1
2	0

Index	Offset
0	0
1	3
2	6
3	10

Offset	Value
0	o
1	l
2	a
3	f
…	…

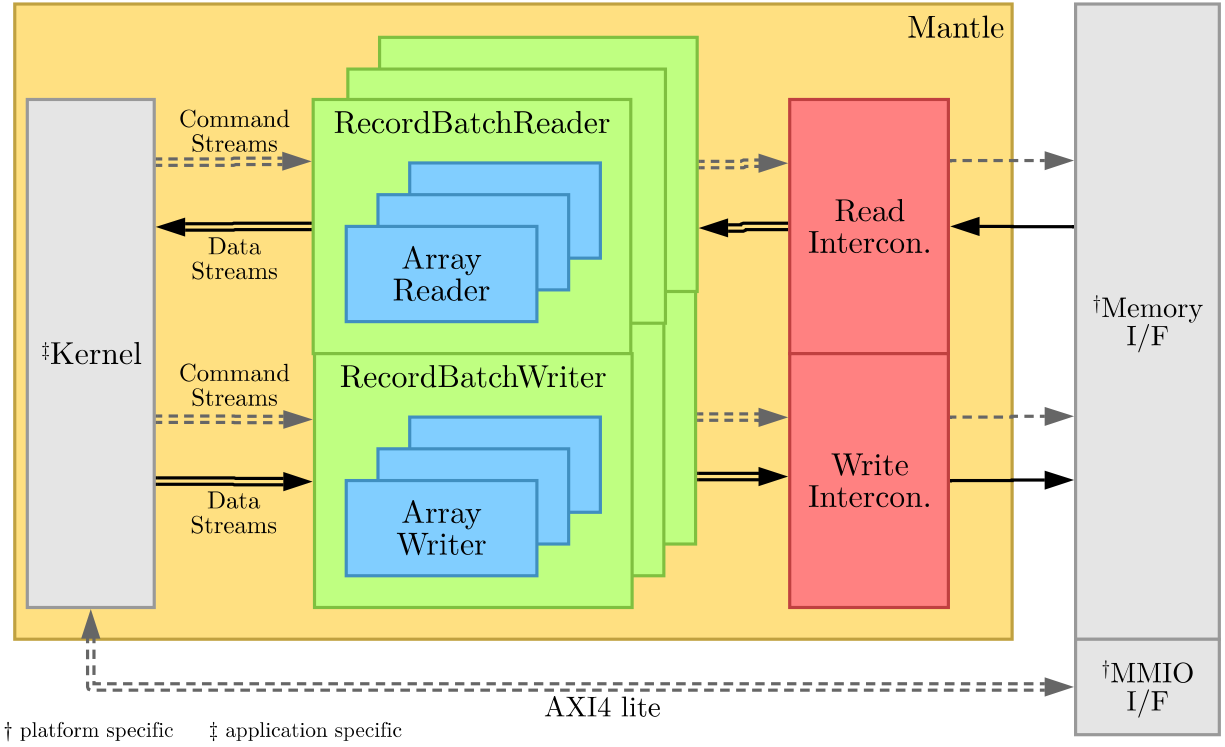
Index	Value
0	1
1	5
2	3

Index	Value
0	3.14
1	1.41
2	1.61

- Arrow Schema & format spec dictate how to combine buffers.
- Passed to ArrayReaders through configuration string in HDL.
- Seeking the limits of synthesis tools :-)
- Over 10k+ random field types simulated.

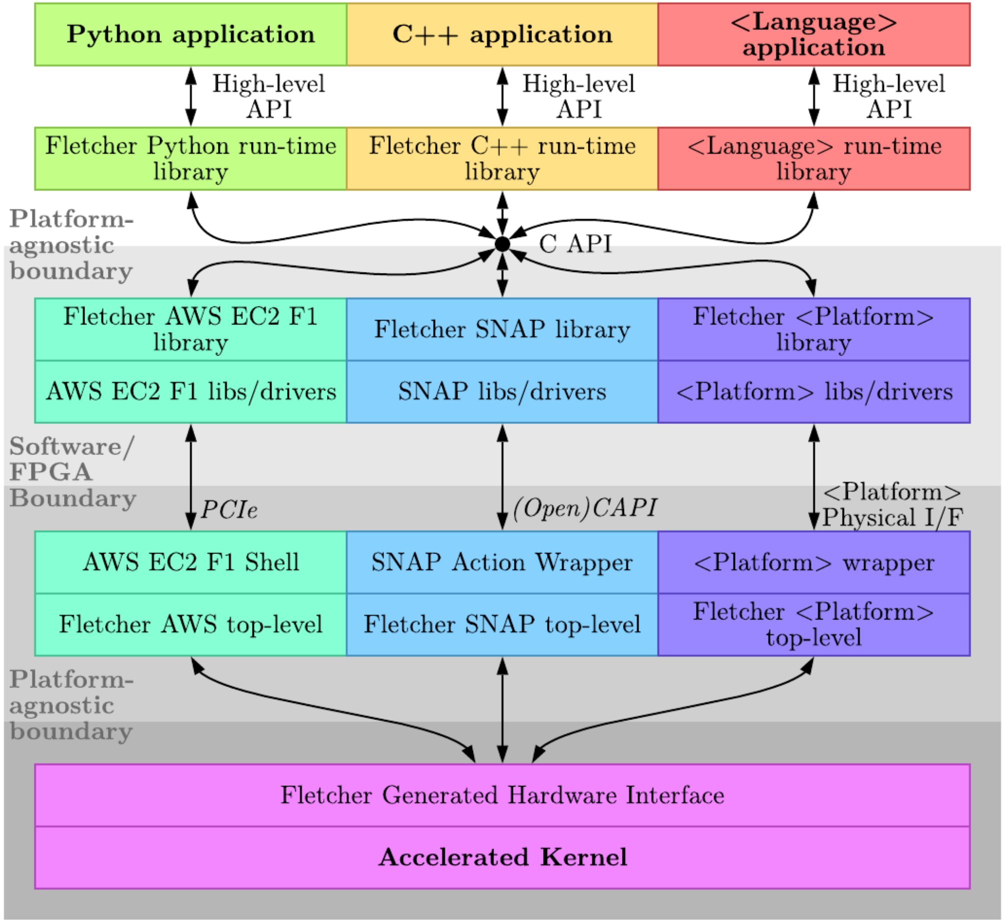
High-level architecture generation: Fletchgen

Arrow support:
☑ **RecordBatches**
☑ Arrays
☑ Buffers



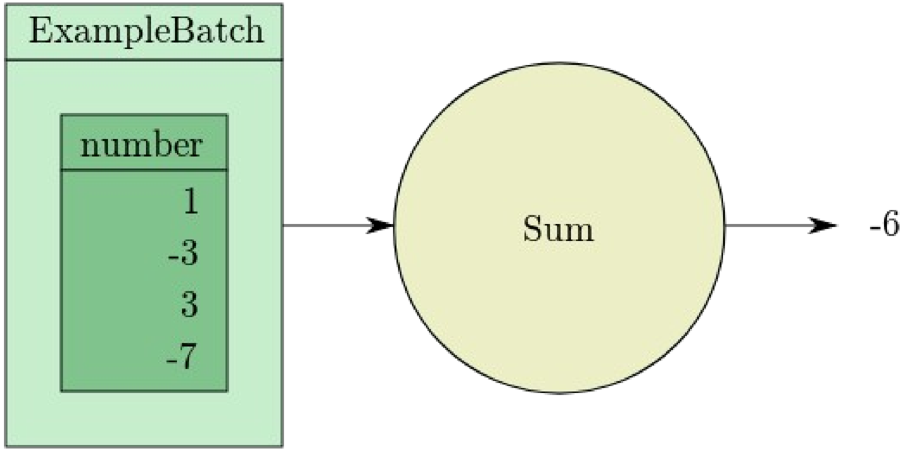
- Need syntactically pleasing interfaces
  - Grouping of ArrayReader/Writer interfaces for RecordBatches
  - Stream names must correspond to schema fields
  - *Synthesizable* HDL too limited
- Need kernel template generation for kernel implementation in HDL/HLS
- Need simulation
- Need platform integration
- High-level architecture generator: Fletchgen

Fletcher run-time stack



- Reap the benefits of Arrow:
  - Create one accelerator.
  - Leverage in any supported language.
- Fletcher Generated Hardware Interface is platform agnostic – requires no IP, tcl scripts, etc…
- Top level with AXI4 interface available.

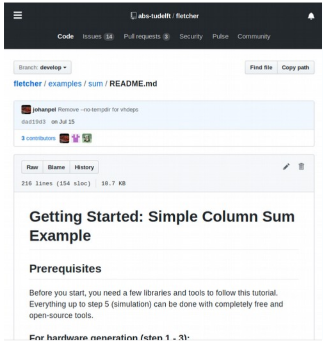
Mini-tutorial: Fletcher “Hello, World!”



number
1
-3
3
-7

- Trivial example:
  - Sum a column of integers
- Get to know the toolchain
- More realistic applications:
  - Complex types
  - More Arrow Arrays
  - More input/output RecordBatches

Also on GitHub:
https://github.com/abs-tudelft/fletcher



fletcher / examples / sum / README.md

johanpel Remove --no-tempdir for vhdeps

dad19d3 on Jul 15

3 contributors

216 lines (154 sloc) | 10.7 KB

Getting Started: Simple Column Sum Example

Prerequisites

Before you start, you need a few libraries and tools to follow this tutorial. Everything up to step 5 (simulation) can be done with completely free and open-source tools.

For hardware generation (step 1 - 3):

Step 1: Create an Arrow Schema

```python
import pyarrow as pa

number_field = pa.field('number', pa.int64(), nullable=False)
schema = pa.schema([number_field])

metadata = {b'fletcher_mode': b'read',
            b'fletcher_name': b'ExampleBatch'}
schema = schema.add_metadata(metadata)
```

Step 2: Create a RecordBatch

(optional, for simulation)

```python
data = [pa.array([1, -3, 3, -7])]
recordbatch = pa.RecordBatch.from_arrays(data, schema)

writer = pa.RecordBatchFileWriter('recordbatch.rb', schema)
writer.write(recordbatch)
writer.close()
```

Step 3: Generate the design

RecordBatch input

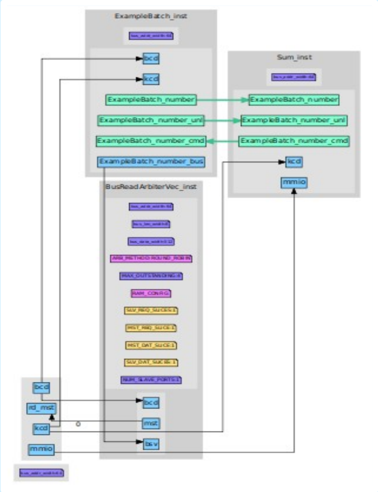
Design output languages

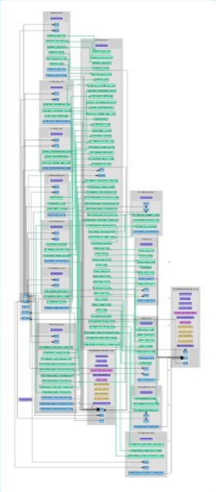
```
$ fletchgen -n Sum -r recordbatch.rb -s recordbatch.srec -l vhdl dot --sim
```

Kernel name

Memory model file

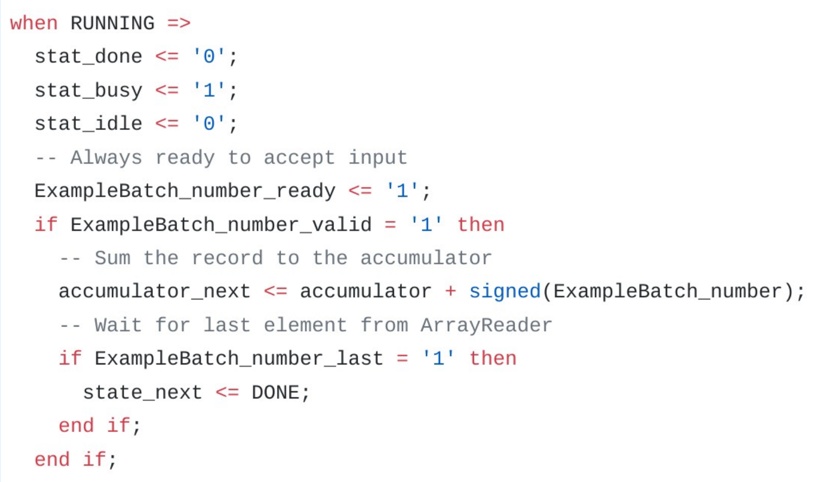
Generate simulation top-level





Step 4: Implement the kernel

- Start from template.
- Use your favorite tools:
  - Custom HDL
  - Vivado HLS
  - …
- Kernel interfaces:
  - AXI4-lite MMIO
  - Command streams to generated interface
  - Data streams from generated interface



```
when RUNNING =>
  stat_done <= '0';
  stat_busy <= '1';
  stat_idle <= '0';
  -- Always ready to accept input
  ExampleBatch_number_ready <= '1';
  if ExampleBatch_number_valid = '1' then
    -- Sum the record to the accumulator
    accumulator_next <= accumulator + signed(ExampleBatch_number);
    -- Wait for last element from ArrayReader
    if ExampleBatch_number_last = '1' then
      state_next <= DONE;
    end if;
  end if;
```

```
int sum(RecordBatchMeta ExampleBatch_meta,
        hls::stream<f_int64>& ExampleBatch_number) {
  ...
}
```

Step 5: Simulate the design

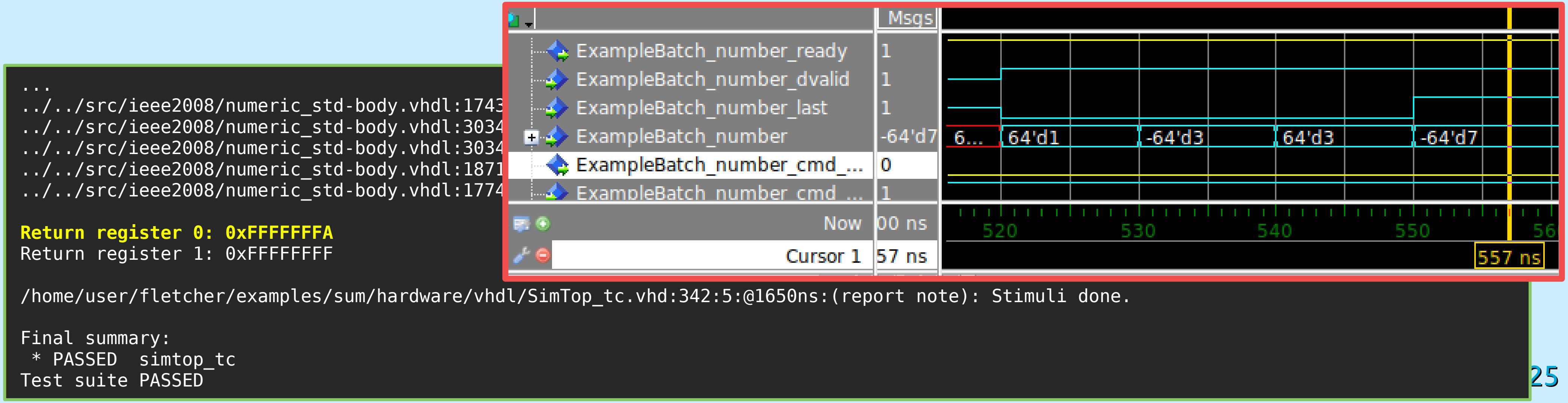
https://github.com/abs-tudelft/vhdeps

Generated simulation top-level

```
$ vhdeps -i path/to/fletcher/hardware -i . ghdl SimTop_tc
```

Fletcher hardware libs

Simulator target
GHDL, Questa, ...



```
...
../../src/ieee2008/numeric_std-body.vhdl:1743
../../src/ieee2008/numeric_std-body.vhdl:3034
../../src/ieee2008/numeric_std-body.vhdl:3034
../../src/ieee2008/numeric_std-body.vhdl:1871
../../src/ieee2008/numeric_std-body.vhdl:1774

Return register 0: 0xFFFFFFFA
Return register 1: 0xFFFFFFFF

/home/user/fletcher/examples/sum/hardware/vhdl/SimTop_tc.vhd:342:5:@1650ns:(report note): Stimuli done.

Final summary:
 * PASSED  simtop_tc
Test suite PASSED
```

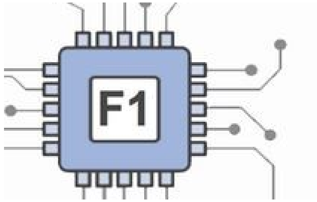
Msgs
ExampleBatch_number_ready 1
ExampleBatch_number_dvalid 1
ExampleBatch_number_last 1
ExampleBatch_number -64'd7
ExampleBatch_number_cmd_... 0
ExampleBatch_number_cmd_... 1
6...
64'd1
-64'd3
64'd3
-64'd7
Now 00 ns
Cursor 1 57 ns
520
530
540
550
56
557 ns

Step 6: Write host-side software

```
import pyarrow as pa
import pyfletcher as pf

...

    platform = pf.Platform()
    platform.init()

    context = pf.Context(platform)
    context.queue_record_batch(batch)
    context.enable()

    kernel = pf.Kernel(context)
    kernel.start()
    kernel.wait_for_finish()


...
```

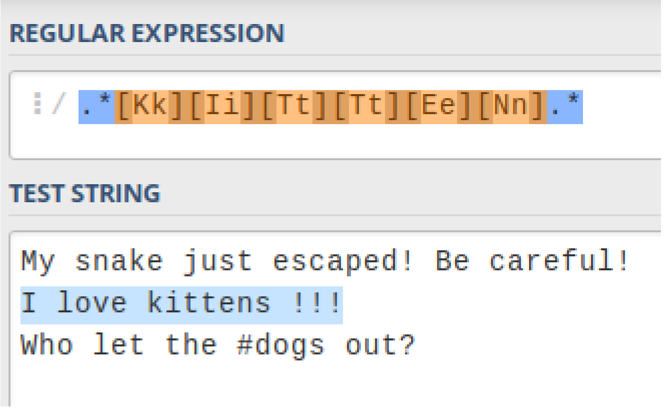
Step 7: Target a Platform

- Supported platforms:
  - OpenPOWER CAPI SNAP
    - If implementation allows, directly streamable from host-memory using virtual addresses on FPGA
  - AWS EC2 F1
    - Requires copy to on-board memory



Regular Expression Matching

- Given N strings
- Match M regular expressions
- Count matches for each regexp
- Example:

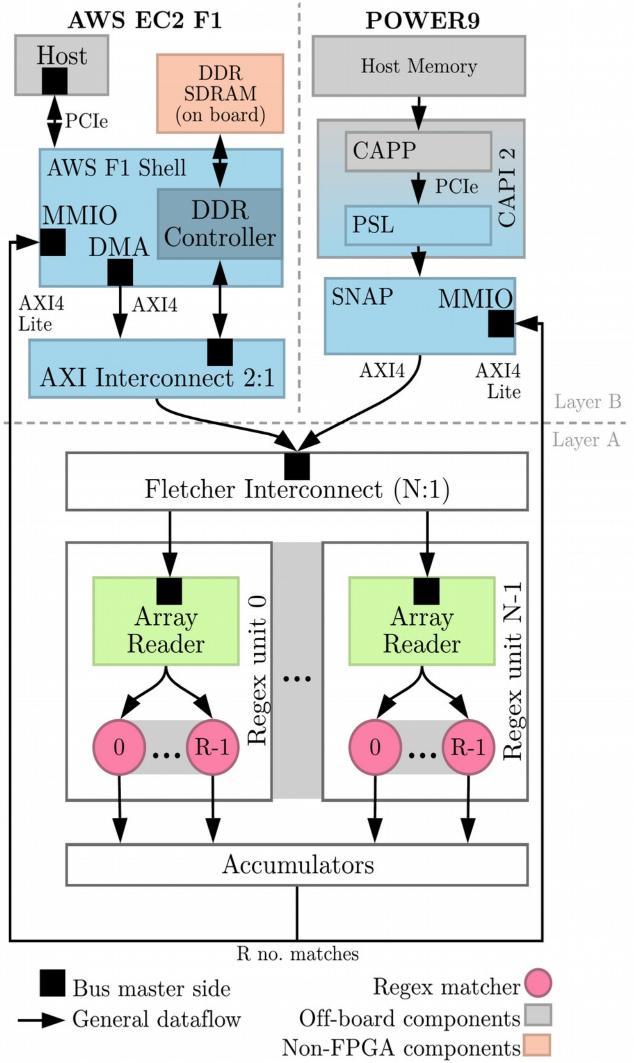


REGULAR EXPRESSION

```
/ .*[Kk][Ii][Tt][Tt][Ee][Nn].*
```

TEST STRING

```
My snake just escaped! Be careful!
I love kittens !!!
Who let the #dogs out?
```



Regex throughput/speedup result

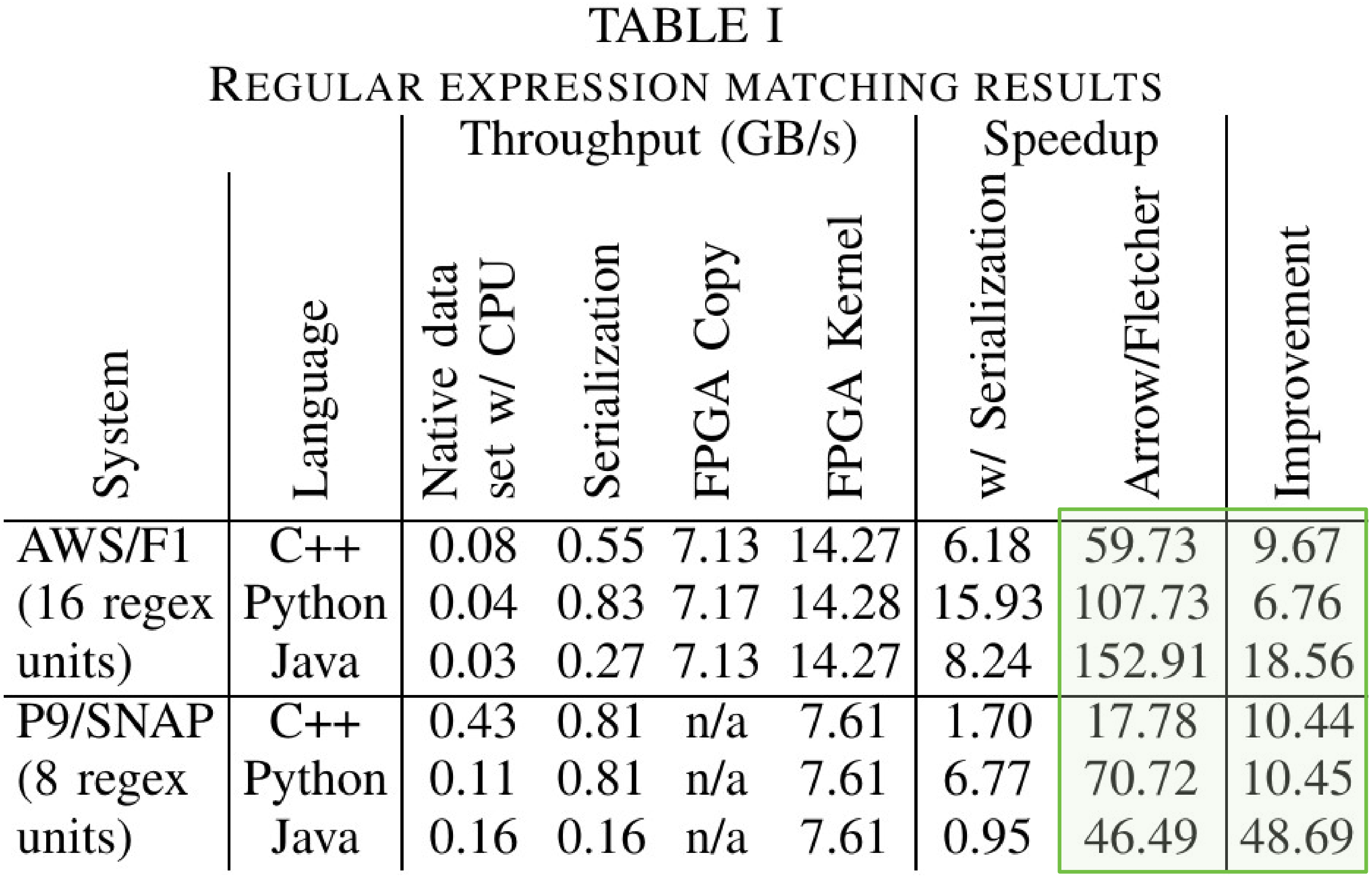
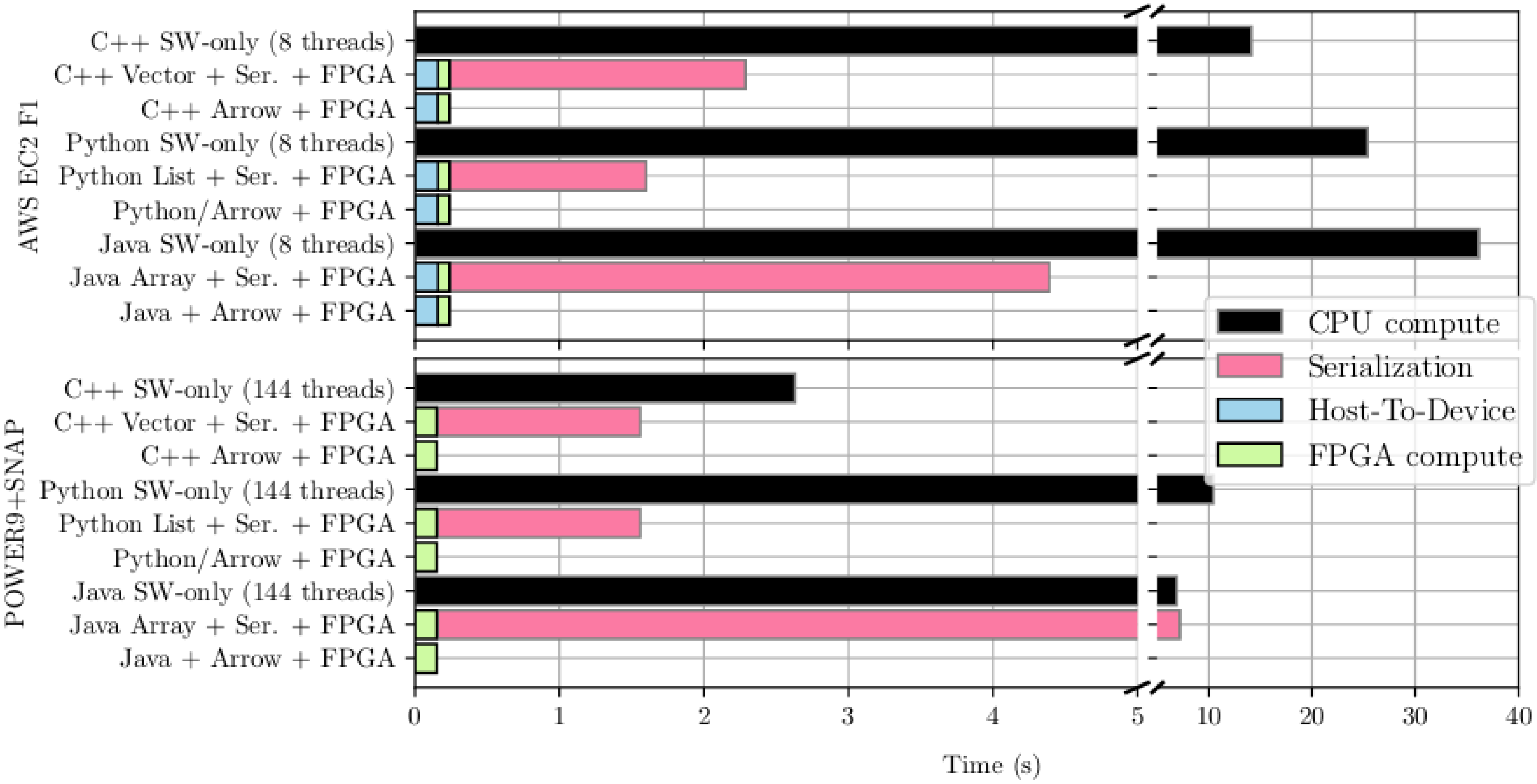


TABLE I

Regular expression matching results

System	Language	Throughput (GB/s)				Speedup		
		Native data set w/ CPU	Serialization	FPGA Copy	FPGA Kernel	w/ Serialization	Arrow/Fletcher	Improvement
AWS/F1 (16 regex units)	C++	0.08	0.55	7.13	14.27	6.18	59.73	9.67
	Python	0.04	0.83	7.17	14.28	15.93	107.73	6.76
	Java	0.03	0.27	7.13	14.27	8.24	152.91	18.56
P9/SNAP (8 regex units)	C++	0.43	0.81	n/a	7.61	1.70	17.78	10.44
	Python	0.11	0.81	n/a	7.61	6.77	70.72	10.45
	Java	0.16	0.16	n/a	7.61	0.95	46.49	48.69

Regex on 1GiB of tweet-like strings



Writing random length (0-255) strings

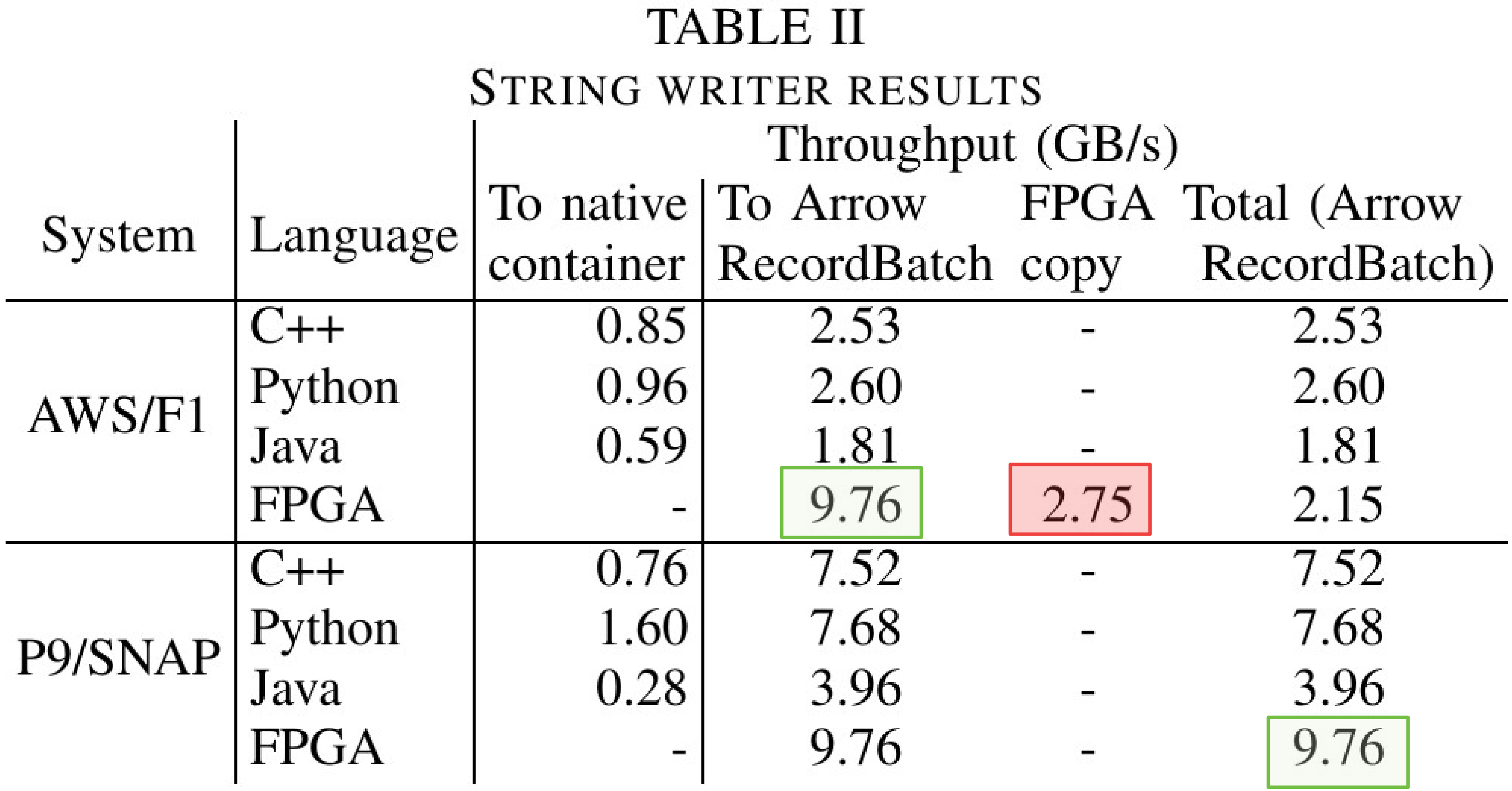


TABLE II

STRING WRITER RESULTS

System	Language	Throughput (GB/s)			
		To native container	To Arrow RecordBatch	FPGA copy	Total (Arrow RecordBatch)
AWS/F1	C++	0.85	2.53	-	2.53
	Python	0.96	2.60	-	2.60
	Java	0.59	1.81	-	1.81
	FPGA	-	9.76	2.75	2.15
P9/SNAP	C++	0.76	7.52	-	7.52
	Python	1.60	7.68	-	7.68
	Java	0.28	3.96	-	3.96
	FPGA	-	9.76	-	9.76

K-Means clustering, internal iteration bandwidth & total run-time

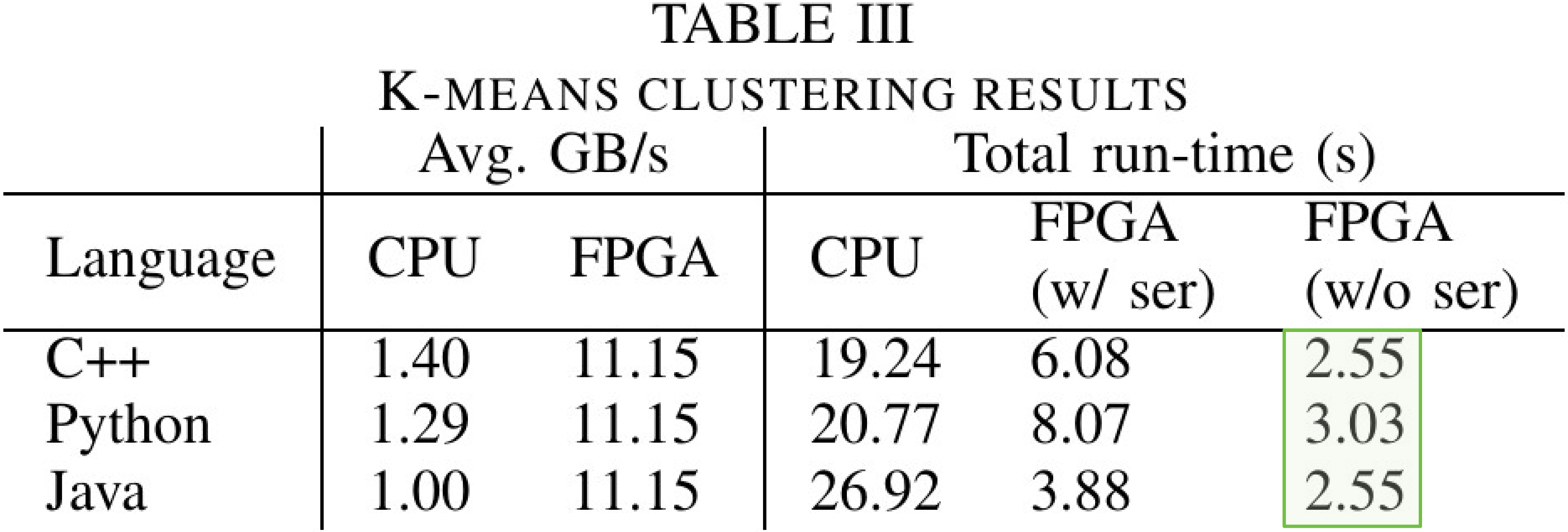


TABLE III

K-MEANS CLUSTERING RESULTS

	Avg. GB/s		Total run-time (s)		
Language	CPU	FPGA	CPU	FPGA (w/ ser)	FPGA (w/o ser)
C++	1.40	11.15	19.24	6.08	2.55
Python	1.29	11.15	20.77	8.07	3.03
Java	1.00	11.15	26.92	3.88	2.55

AWS EC2 F1 only

Conclusion

- Big data analytics systems increasingly heterogeneous – many different tools in many different technologies.
- Apache Arrow: one in-memory format for IPC through shared memory for most languages / runtimes / technologies.
- Fletcher: Arrow format allows us to generate high-throughput, easy-to-use hardware interfaces for FPGA.
- Streaming kernels benefit the most, more computationally oriented kernels less.
- Paves the way for more efficient FPGA accelerator integration with any of the supported big data analytics tools.

Spin-off projects & future work

- **Dynamic Arrow Buffers in Hardware** to support buffer resizing (Lars Wijtemans)
- **Parquet-to-Arrow decoder and decompressor** (Lars van Leeuwen, Jian Fang)
- HLS integration for map, reduce, **SQL-defined filters** (Erwin de Haan)

- Data-defined architecture.
- Can we turn this into a closed-loop self-optimizing interface generation and profiling framework?
  - Long-running workload: plenty of time to synthesize.
  - We only need one node of a cluster to do it.
  - Are the gains worth the cost?

Thank you for your attention!

References:

[1] F. Kruger, “CPU Bandwidth The Worrisome 2020 Trend,” Mar. 2016. [Online]. Available: https://blog.westerndial.com/cpu-bandwidth-the-worrisome-2020-trend/

[2] J. Peltenburg, A. Hesam, and Z. Al-Ars, “**Pushing Big Data into Accelerators: Can the JVM Saturate Our Hardware?**” in International Conference on High Performance Computing. Springer, 2017, pp. 220–236.

[3] The Apache Software Foundation, “Apache Arrow,” 2018. [Online]. Available: https://arrow.apache.org

[4] J. Peltenburg, J. van Straten, M. Brobbel, H. P. Hofstee, and Z. Al-Ars, “**Supporting Columnar In-memory Formats on FPGA: The Hardware Design of Fletcher for Apache Arrow**”, in Applied Reconfigurable Computing, Cham: Springer International Publishing, 2019, pp. 32–47.

FLETCHER

https://github.com/abs-tudelft/fletcher

Open-sourced example projects / existing applications:

- Regular Expression matching example / benchmark:
https://github.com/abs-tudelft/fletcher-example-regexp
- K-Means clustering example/benchmark:
https://github.com/abs-tudelft/fletcher-example-kmeans
- Writing strings to Arrow format using CAPI 2.0 and SNAP @ 10 GB/s:
https://github.com/abs-tudelft/fletcher/blob/develop/examples/stringwrite
- Posit arithmetic on FPGA, BLAS and PairHMM accelerators by Laurens van Dam:
https://github.com/lvandam/posit_blas_hdl
https://github.com/lvandam/pairhmm_posit_hdl_arrow

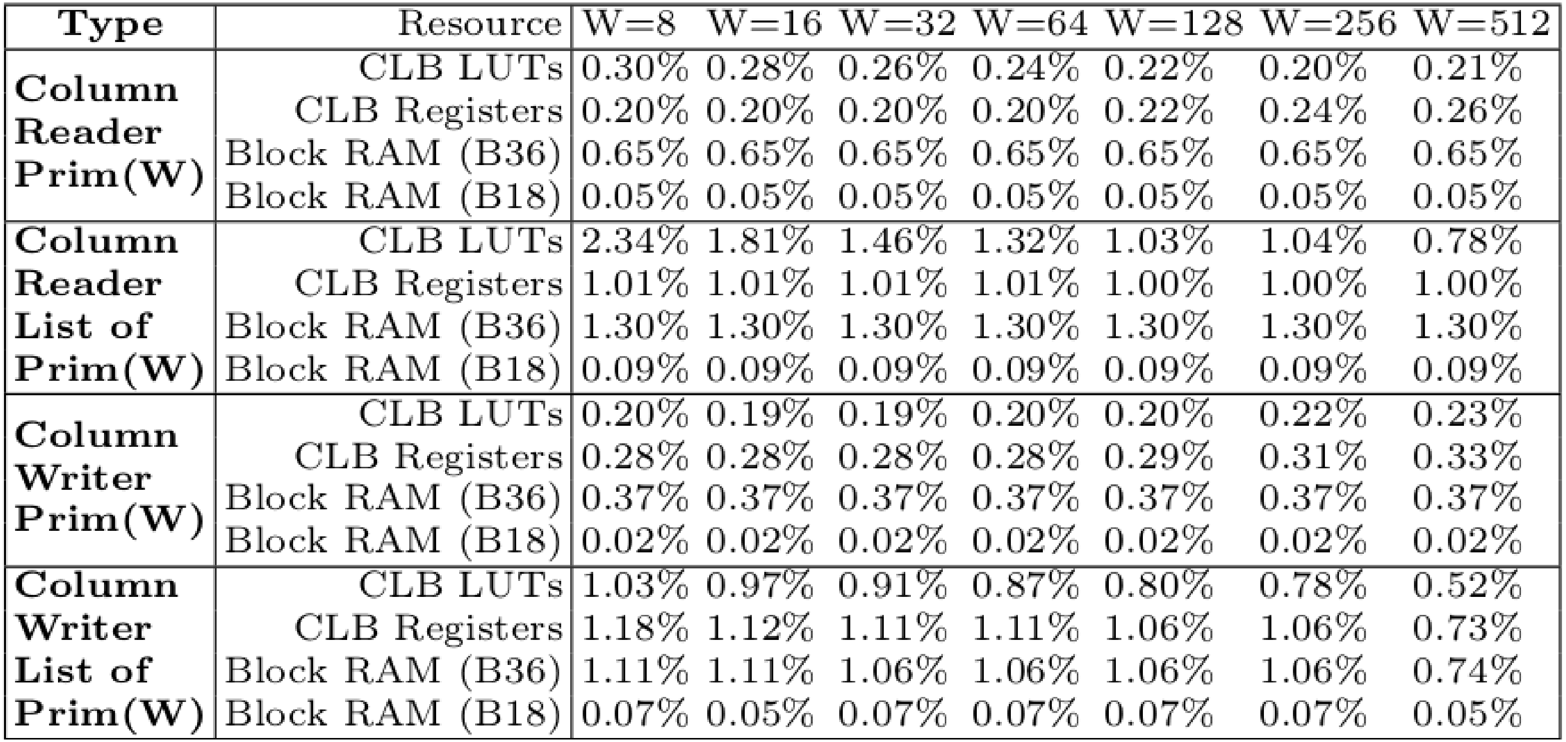
Thanks to our supporters:

Fitoptivis European ECSEL project
no. ECSEL2017-1-737451

Xilinx

Area utilization

Tab. 2: Area utilization statistics for a Xilinx XCVU9P device



Type	Resource	W=8	W=16	W=32	W=64	W=128	W=256	W=512
Column Reader Prim(W)	CLB LUTs	0.30%	0.28%	0.26%	0.24%	0.22%	0.20%	0.21%
	CLB Registers	0.20%	0.20%	0.20%	0.20%	0.22%	0.24%	0.26%
	Block RAM (B36)	0.65%	0.65%	0.65%	0.65%	0.65%	0.65%	0.65%
	Block RAM (B18)	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%	0.05%
Column Reader List of Prim(W)	CLB LUTs	2.34%	1.81%	1.46%	1.32%	1.03%	1.04%	0.78%
	CLB Registers	1.01%	1.01%	1.01%	1.01%	1.00%	1.00%	1.00%
	Block RAM (B36)	1.30%	1.30%	1.30%	1.30%	1.30%	1.30%	1.30%
	Block RAM (B18)	0.09%	0.09%	0.09%	0.09%	0.09%	0.09%	0.09%
Column Writer Prim(W)	CLB LUTs	0.20%	0.19%	0.19%	0.20%	0.20%	0.22%	0.23%
	CLB Registers	0.28%	0.28%	0.28%	0.28%	0.29%	0.31%	0.33%
	Block RAM (B36)	0.37%	0.37%	0.37%	0.37%	0.37%	0.37%	0.37%
	Block RAM (B18)	0.02%	0.02%	0.02%	0.02%	0.02%	0.02%	0.02%
Column Writer List of Prim(W)	CLB LUTs	1.03%	0.97%	0.91%	0.87%	0.80%	0.78%	0.52%
	CLB Registers	1.18%	1.12%	1.11%	1.11%	1.06%	1.06%	0.73%
	Block RAM (B36)	1.11%	1.11%	1.06%	1.06%	1.06%	1.06%	0.74%
	Block RAM (B18)	0.07%	0.05%	0.07%	0.07%	0.07%	0.07%	0.05%