EFFICIENT DATA INGESTION

March 27th 2018

FASTDATA.IO

Data Processing at the Speed of Thought

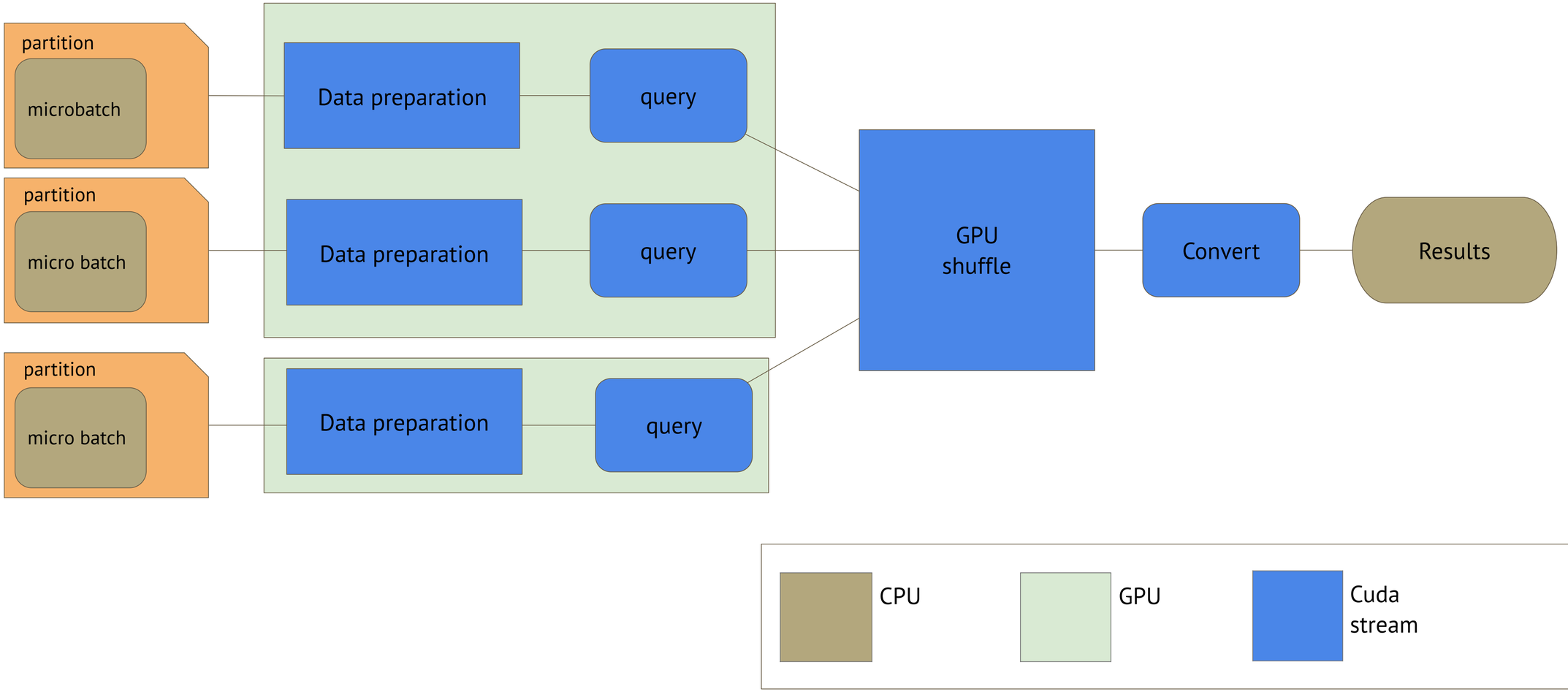
Performance Goals

- Must be limited to hardware constraint
- Disk, Network and PCI bus are the most critical
- Speeds up to 10GB/s per GPU are theoretically possible
- Conservative goal: 2-4GB/s per GPU card for complex queries
- With CPU only approach we get 10 MB/s per core
- GPU executor speed up of 200-400 times

Problem Statement

- GPU Accelerated large-scale structured data processing
- Similar stream and batch programming model
- Treat microbatches as temporary database tables
- Existing GPU Databases demonstrated that queries work very fast
- They do not solve the problem of very fast data loading due to the CPU bottleneck
- Data Ingestion Problem

GPU Streaming Architecture

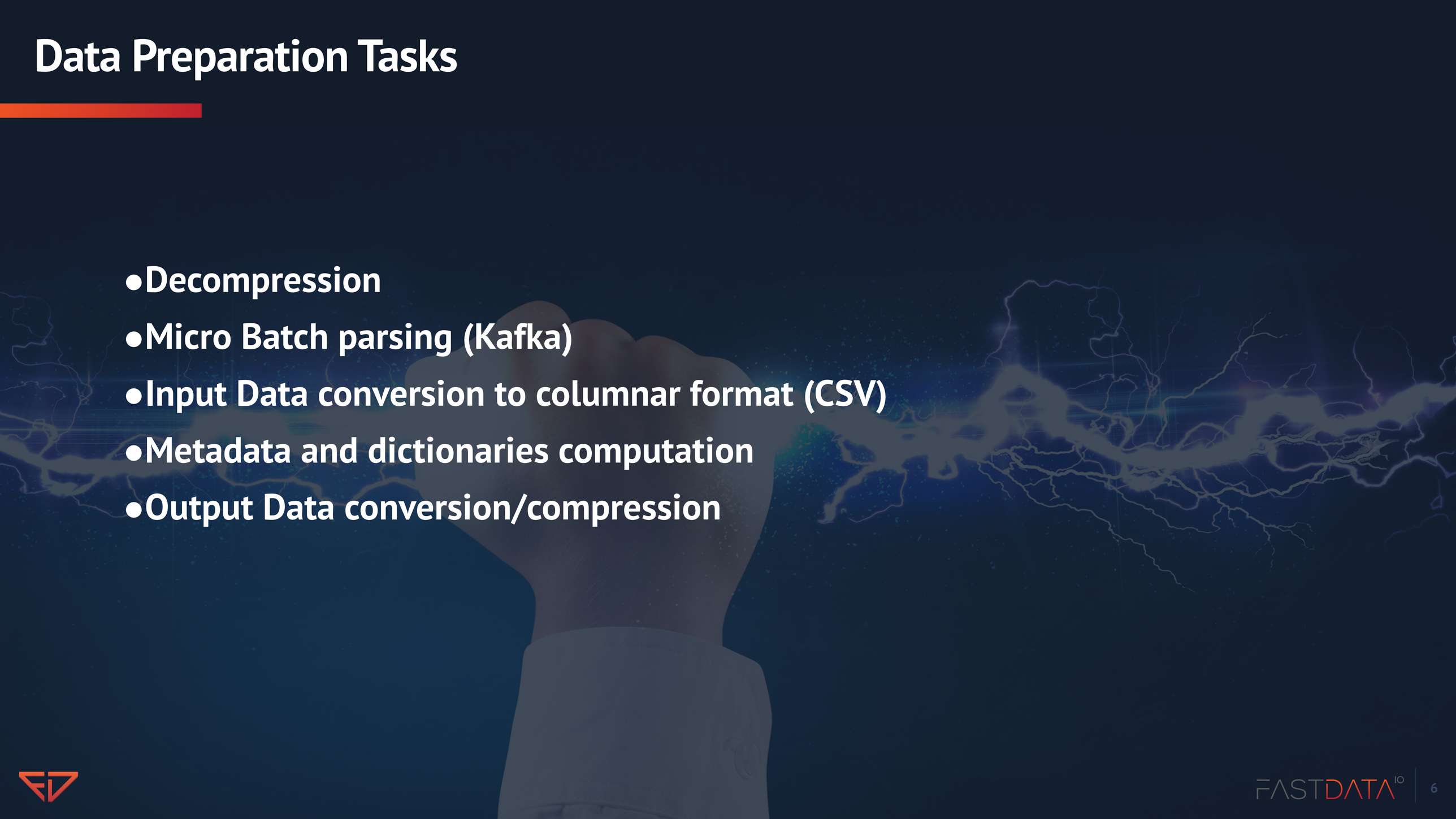


Problem Constraint Mitigation Strategies

Constraints	Speed	Strategies
CPU	-	Data preparation on GPU
Disk	2 GB/s * new NVMe disks	Compression, partitions
Network	3 GB/s	Compression, partitions
PCI Bus	10 GB/s	NVLINK (only on Power)
RAM	up to 60 GB/s	GpuDirect RDMA
VRAM	up to 1 TB/s * Assumes optimal memory access by threads	Columnar format, shared memory

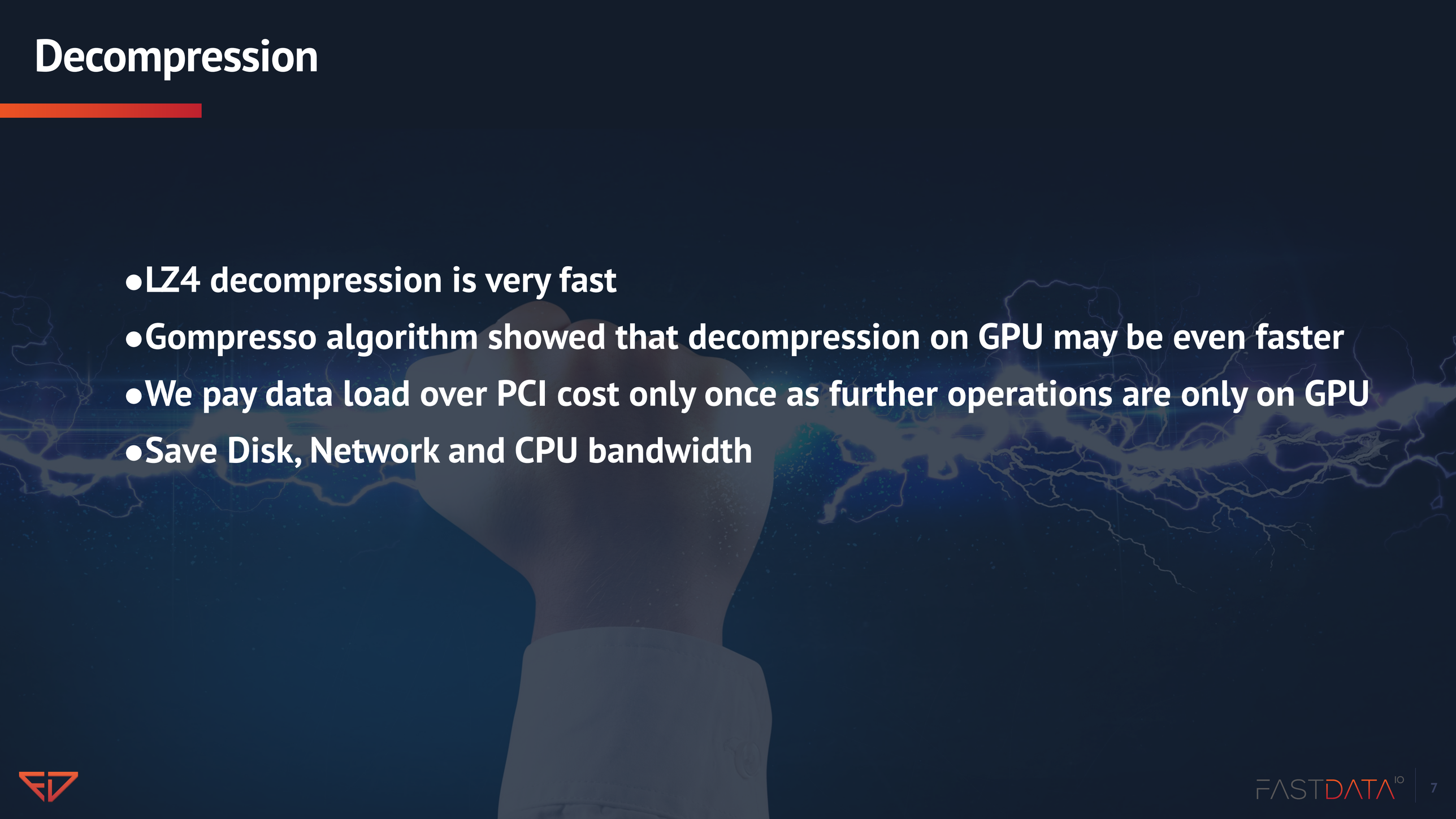
Data Preparation Tasks

- Decompression
- Micro Batch parsing (Kafka)
- Input Data conversion to columnar format (CSV)
- Metadata and dictionaries computation
- Output Data conversion/compression



Decompression

- LZ4 decompression is very fast
- Gompresso algorithm showed that decompression on GPU may be even faster
- We pay data load over PCI cost only once as further operations are only on GPU
- Save Disk, Network and CPU bandwidth



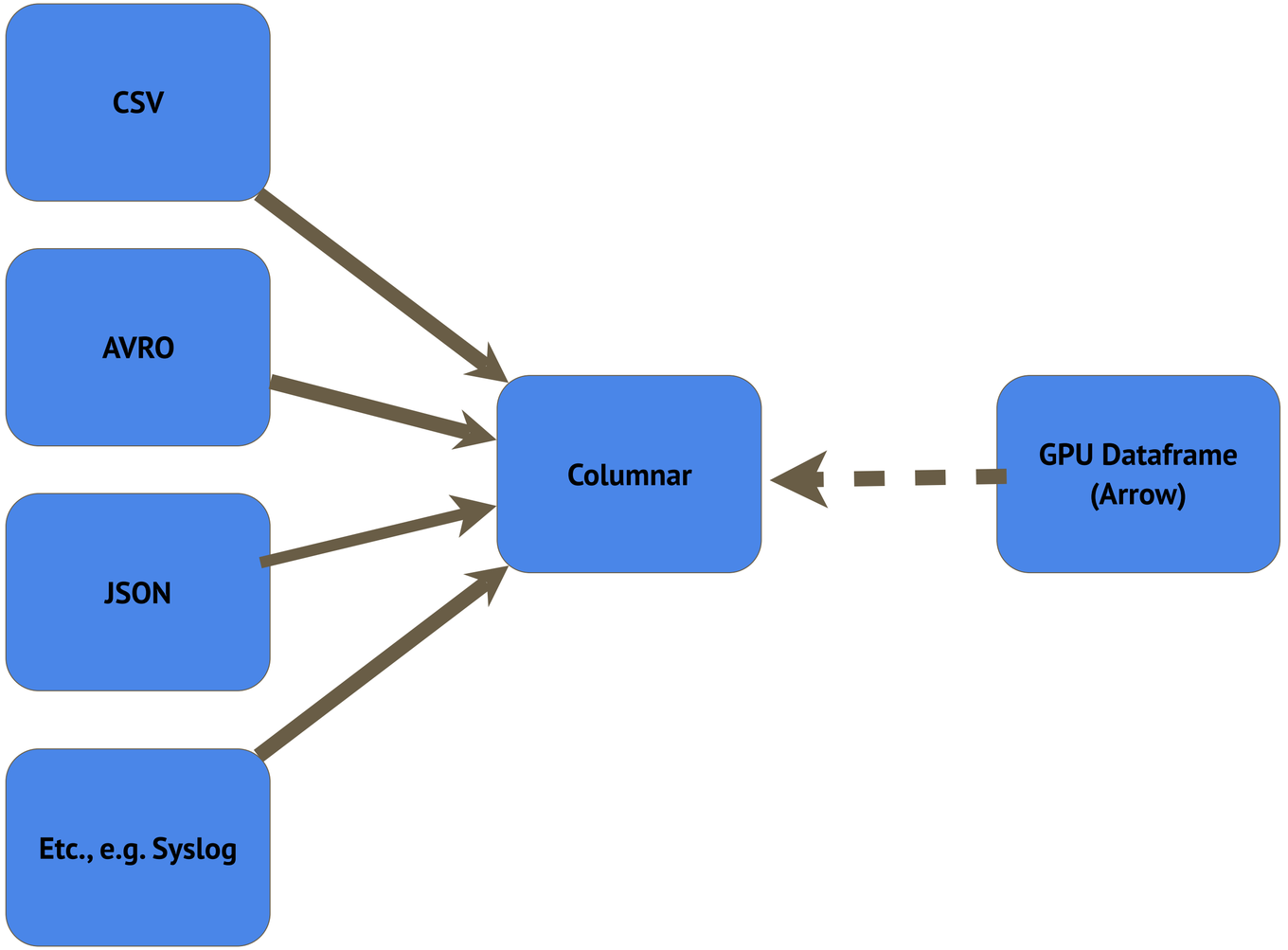
Micro Batch Parsing

1) Kafka interface gives random access to messages by message index
2) Each message may have arbitrary format, e.g. CSV line
3) Allows batch read of multiple messages
4) Spark Kafka client parses the batch and generates a Java object for each message
5) We still need to pass batch to GPU

Solution: skip Java object generation and pass raw message batch to GPU

Result: 2.5-3x speedup

Input Data Conversion to Columnar Format



Metadata and Dictionaries

1) Dictionary is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.
2) Need two vectorized operations:
   a) Insert string
   b) Lookup string
3) The two major solutions to the dictionary problem are a hash table or a search tree. Not optimal on GPU
4) We aim to process up to 4GB/s on a single GPU card and dictionary construction should take not more than ⅓ of that.
5) Our target is at least 12GB/s
6) Dictionaries are required only for string columns and specific queries

Practical Use Case

Input table		
id	time	name
1	00:01	click
2	00:02	view
3	00:03	click

Dictionary	
1	click
2	view

Output table	
name	count
1	2
2	1

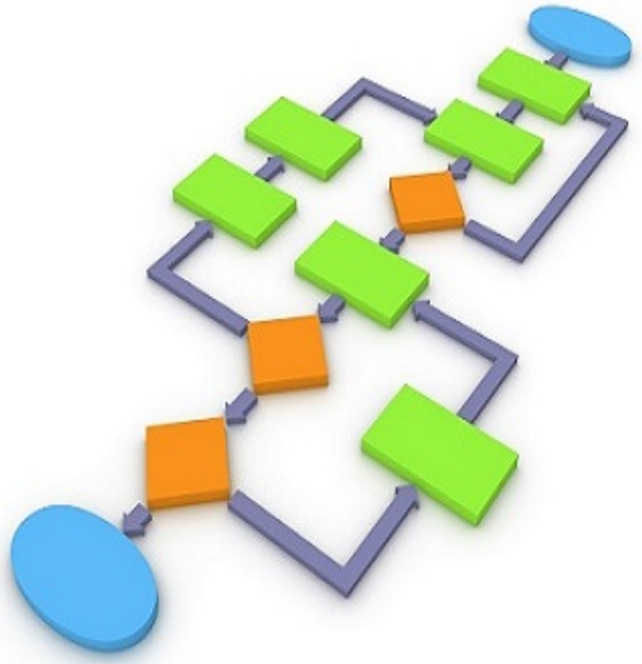
1
2
1

click
view

SELECT NAME, COUNT(NAME) FROM INPUT_TABLE GROUP_BY NAME;

Solutions for Dictionary Structure

Algo	Pros	Cons
Hash table Search tree	Classic solution	Not optimal on GPU
String sort	No collisions	Slow on GPU
Hashes sort	Fast speed on GPU	Collisions



Hash Collisions

- Birthday Paradox
- Probability of 2 people having a birthday on the same day is
  - P = n^2/2m, where n - number of people, m - number of days in the year
- n = sqrt(2m*p)
- For 64 bit hash m = 2^64
- To get a collision with 99.99% probability it would take:
  - N = sqrt(2*2^64*0.9999) = 8 billion strings
- Assuming our engine processes 100 million events per second
- It would have one collision every 80 seconds!
  - 320 billion seconds for 128 bit
- Conclusion: even with 64 bit hash, we must handle collisions

Dictionary Construction Algorithm

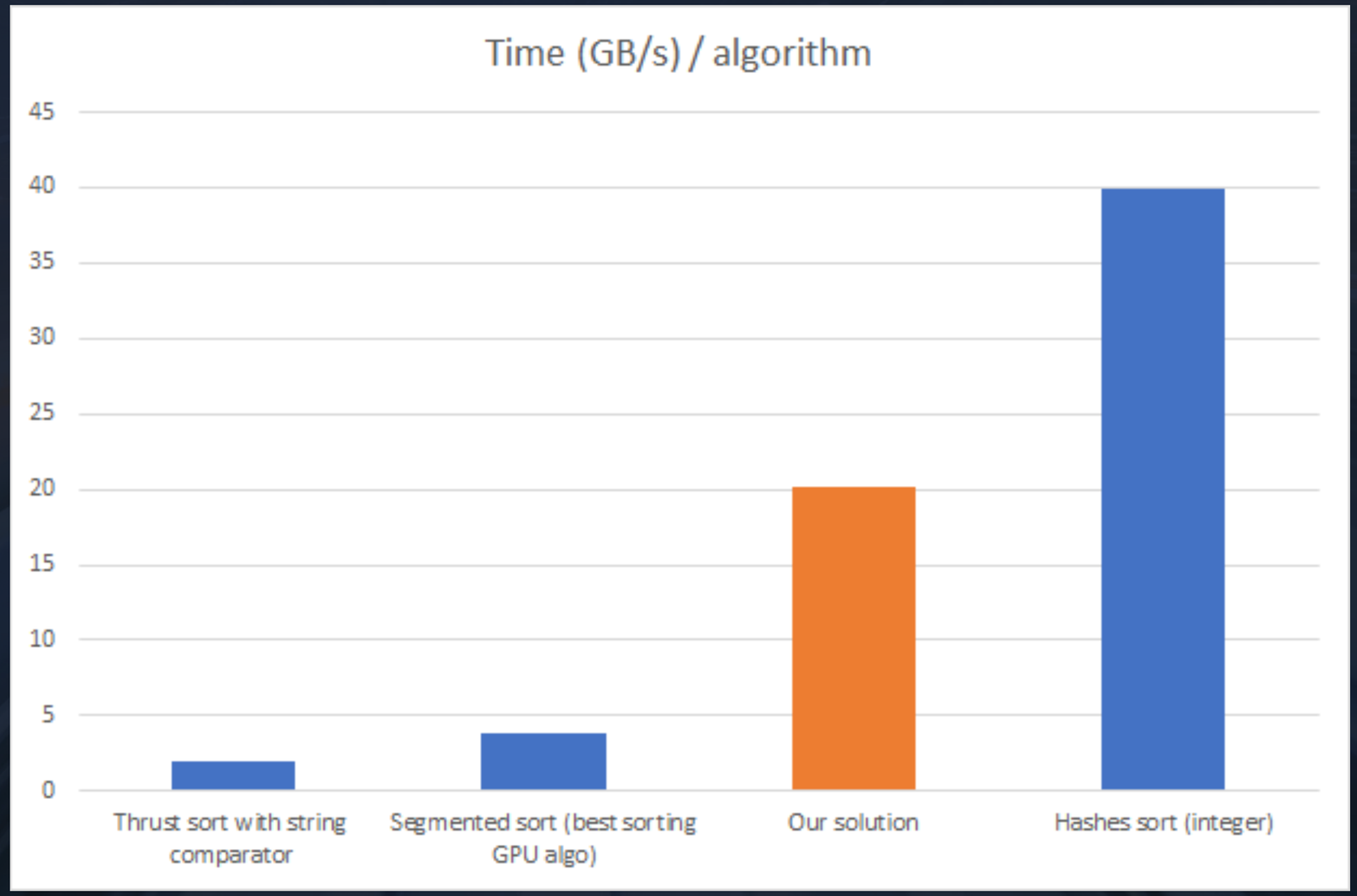
Insert strings

1) Compute hashes for all strings
2) Sort on hash + string key
3) “Unique” operation on hash + string key

Lookup strings

1) Binary search using hash + string key

Comparison of Dictionary Implementation



* 1 mil of 40 byte uuids

* tested on V100

Latest Results

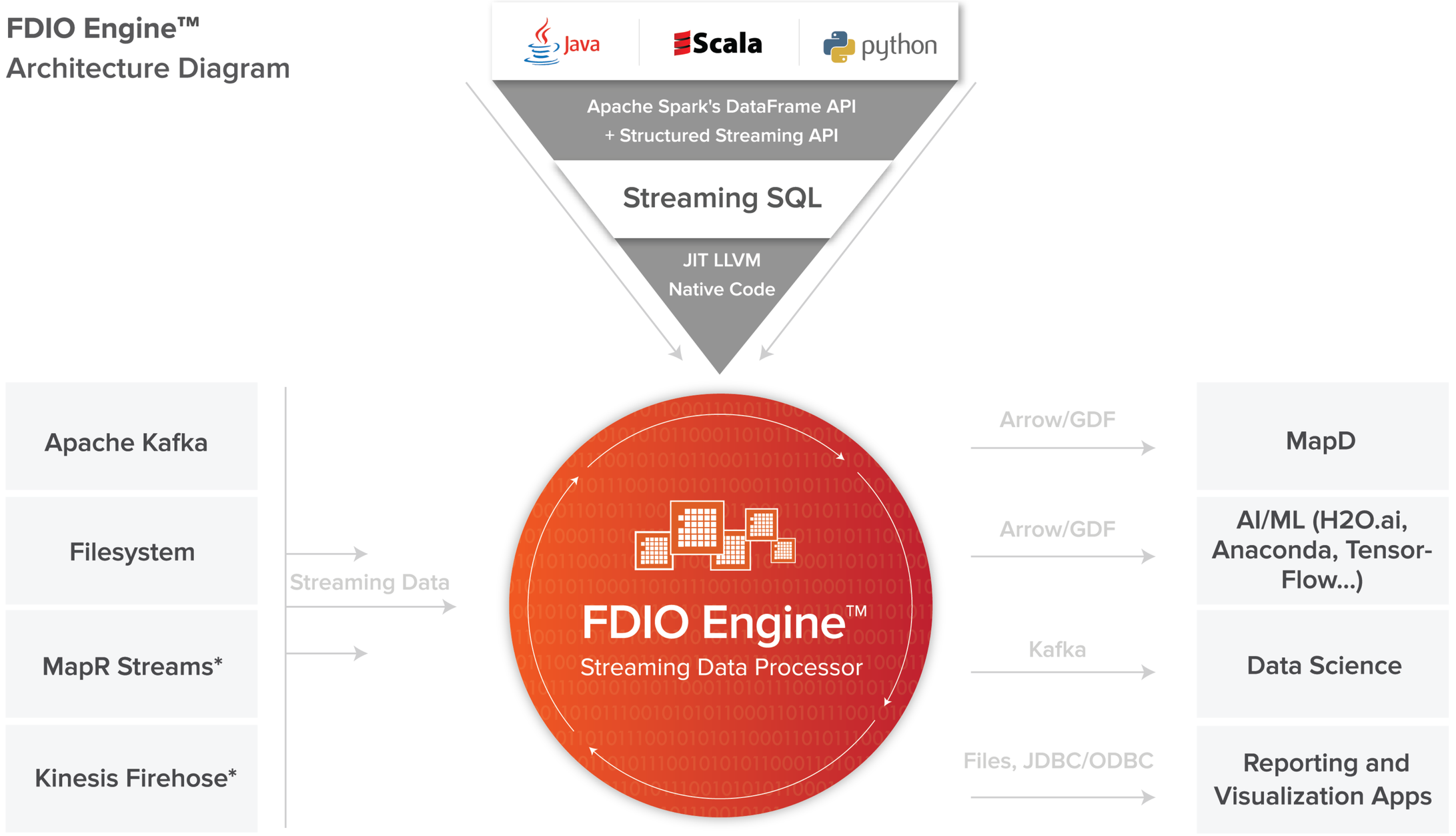
FDIO (one GPU) speed: 1 GB/s (as of now, still improving it)

Original Spark (8 CPU Cores) speed: 60 MB/s

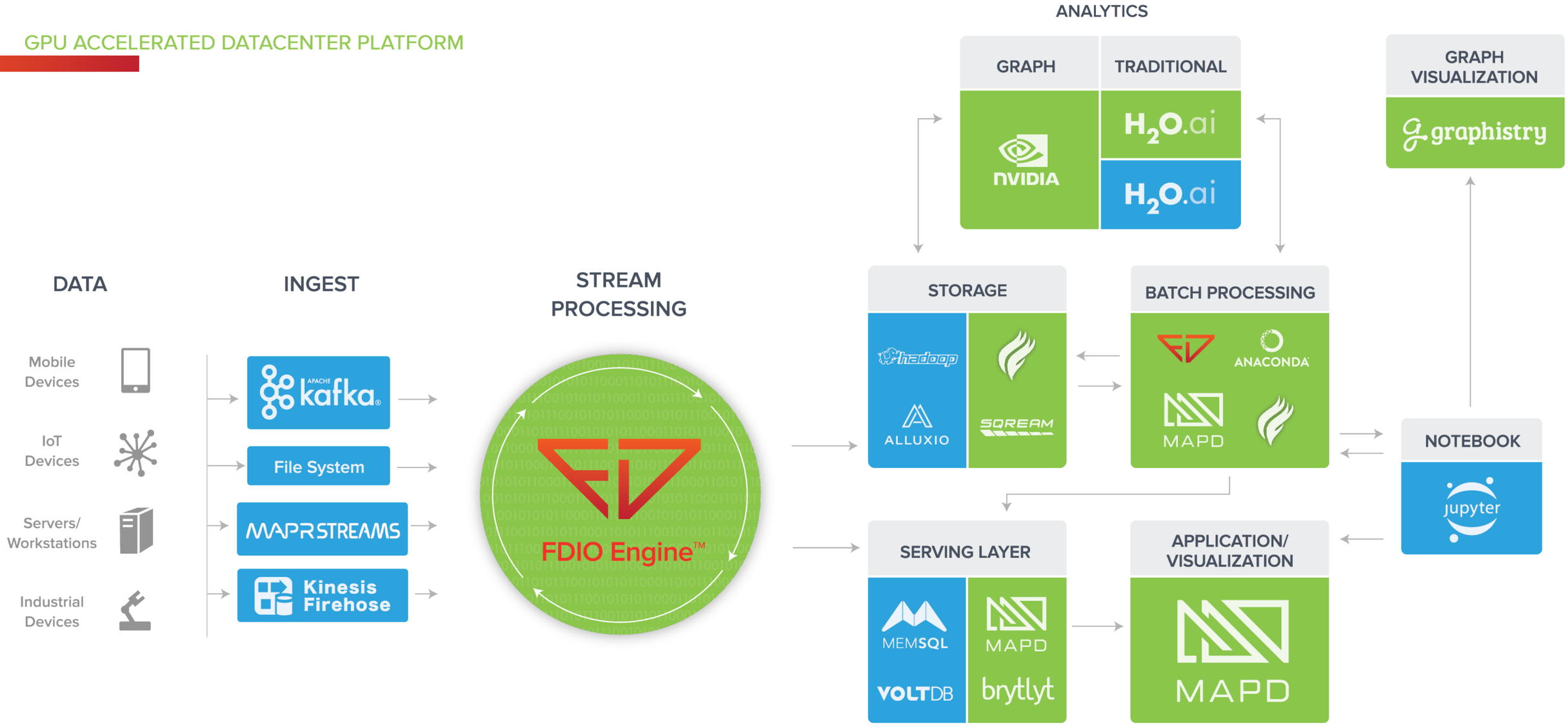
Query on Kafka stream for CSV data aggregation:

```
telecomStream
 .withWatermark("call_time", "60 seconds")
 .groupBy(window($"call_time", "60 seconds"),
$"cell_from")
 .agg(count("*"))
 .join(cellsStaticDf, telecomStream.col("cell_from") ===
cellsStaticDf.col("cell"))
```

FDIO Engine™
Architecture Diagram



GPU ACCELERATED DATACENTER PLATFORM



GPU BASED TECHNOLOGIES

CPU BASED TECHNOLOGIES

Conclusion

- Streaming at gigabytes/sec speed is a very hard problem
- The whole system is only as fast as the slowest link
- What we have is still a target rich environment
- Plenty of optimization opportunities remain
- Still at the beginning of a long journey
- The possibilities are very exciting
- We plan the GA release of FDIO engine in April 2018

Links

- Can GPUs Sort Strings Efficiently?
  http://web.engr.illinois.edu/~ardeshp2/papers/Aditya13StringSort.pdf

- Massively-Parallel Lossless Data Decompression
  https://arxiv.org/pdf/1606.00519

Sign up for a Test Flight POC at fastdata.io

Visit us in the Inception Startup Pavilion at booth 935

FASTDATA.IO

Data Processing at the Speed of Thought

Vassili Gorshkov, CTO

vassili@fastdata.io

1-888-707-3346