Systems Support for Data Analytics by Exploiting Modern Hardware

Hongyu Miao

Purdue ECE

11/15/2021

Committee: Felix Xiaozhu Lin (chair), Kathryn S. McKinley, Mithuna S. Thottethodi, Y. Charlie Hu

Data analytics has become one of the most important workloads, due to data explosion

- 2.5 quintillion bytes/day: e.g., 0.5 million Tweets, 4 PB data on Facebook
- Extracting useful insights from such large volume of data requires *high throughput, low latency, and high accuracy*

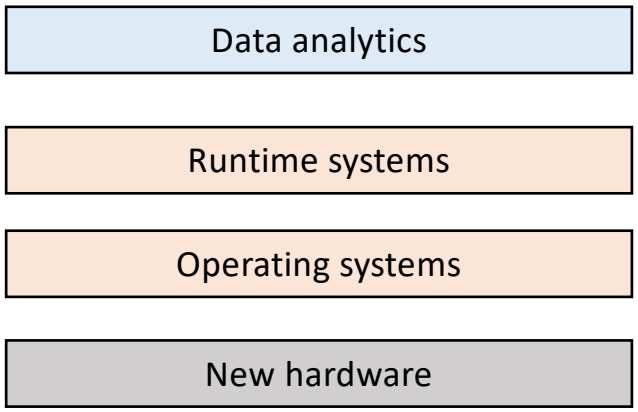
System software (runtime/OS):
Fail to deliver performance requirements due to quickly evolving new hardware and data explosion

Runtime systems:

- Not aware of new HW features
- Poor performance/efficiency

Operating systems

- Not aware of unique demands of data analytics
- Incapable of delivering desired performance



This thesis aims to

- Bridge the gap between data explosion and hardware evolving
- Build systems to improve performance or enable new use cases for data analytics by exploiting modern hardware features
- (designs/optimization across application, algorithm, runtime, OS, and hardware)

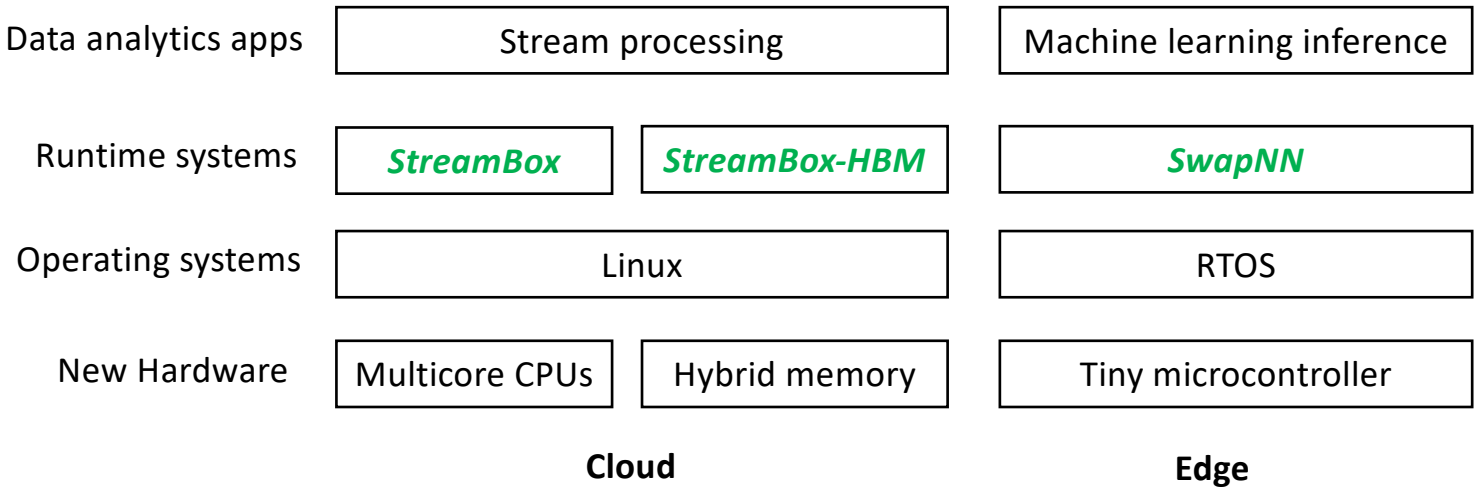
New hardware is emerging, aiming to meet the demands of data analytics

- many-core CPUs for higher computation speed
- 3D-stacked memory for higher memory bandwidth
- Tiny microcontrollers for better energy efficiency

The Theme of This Thesis

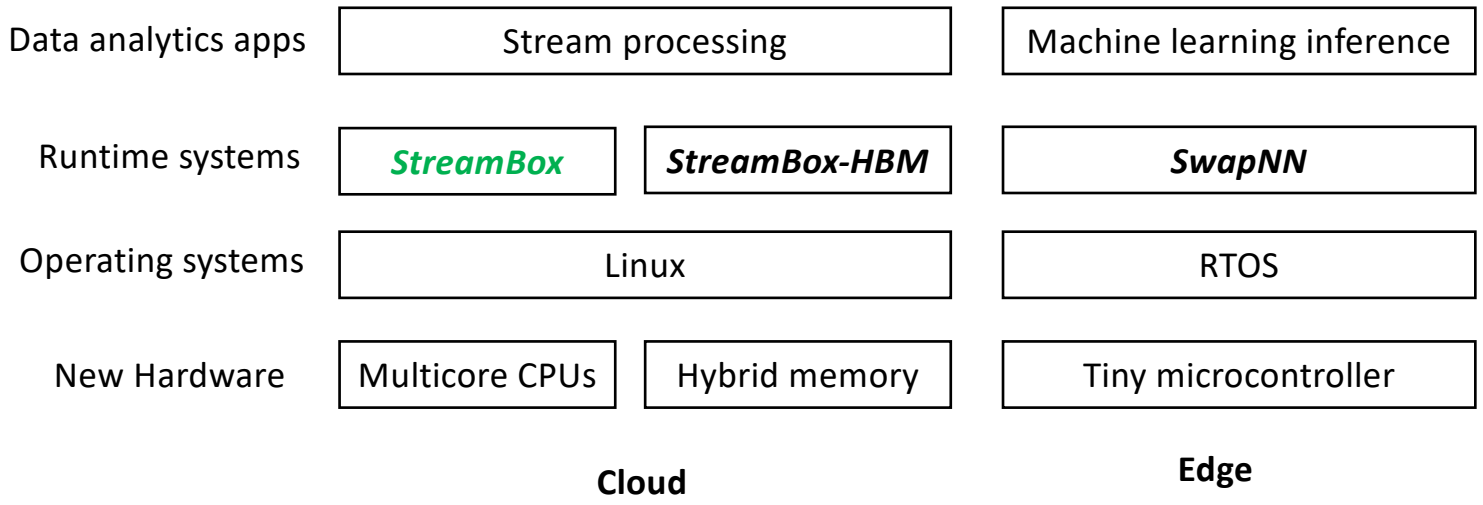
Runtime systems support for data analytics by exploiting modern hardware

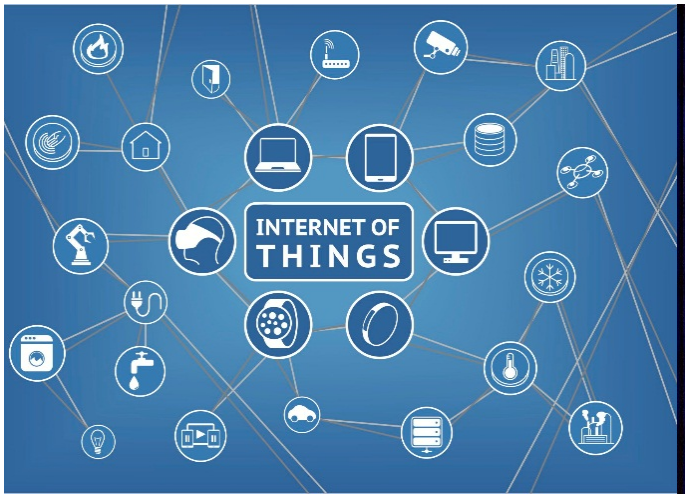
- StreamBox/-HBM: To speed up stream processing on single machine with multicore and hybrid memory
- SwapNN: To enable high-accuracy NN inference on resource-constraint microcontrollers



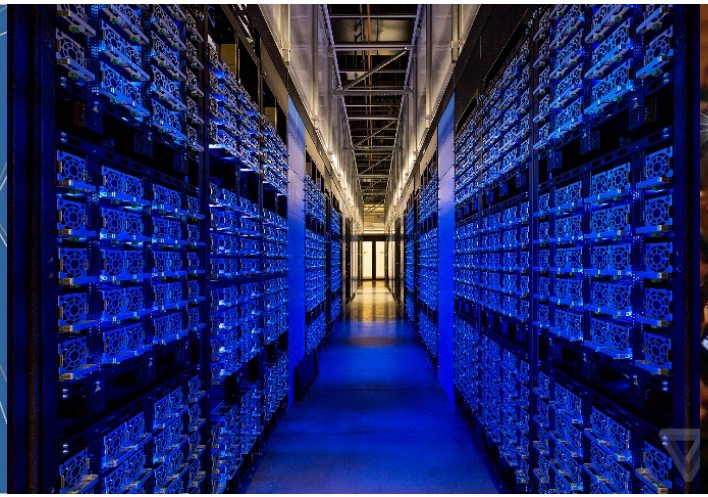
This Thesis: StreamBox

- We exploit the parallelism and memory hierarchy of modern multicore hardware on single machines for stream processing, achieving scalable and highly efficient performance
- Key results on single node: reduce 20x latency, improve throughput by an order of magnitude





IoT



Data centers

Humans

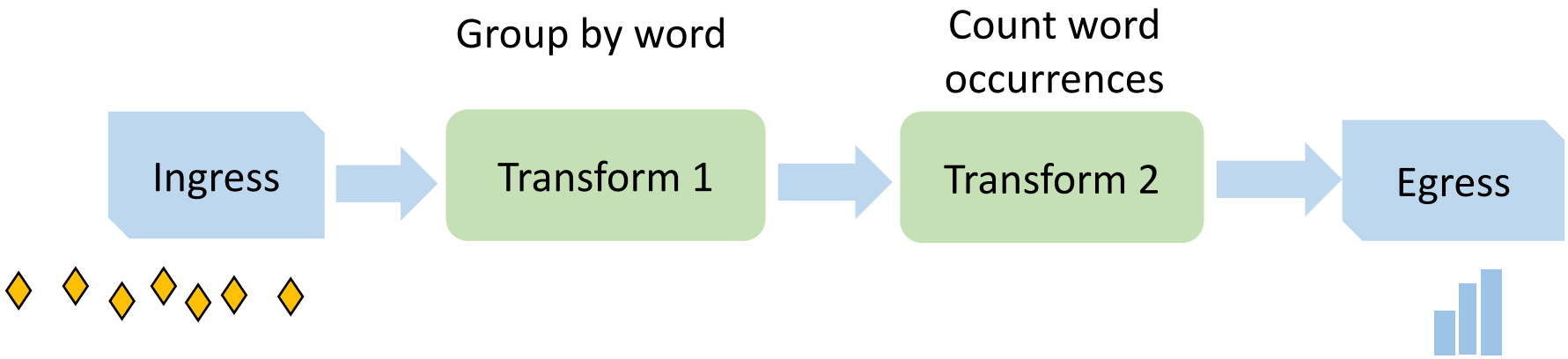
High velocity of streaming data

requires real-time processing

Streaming pipeline

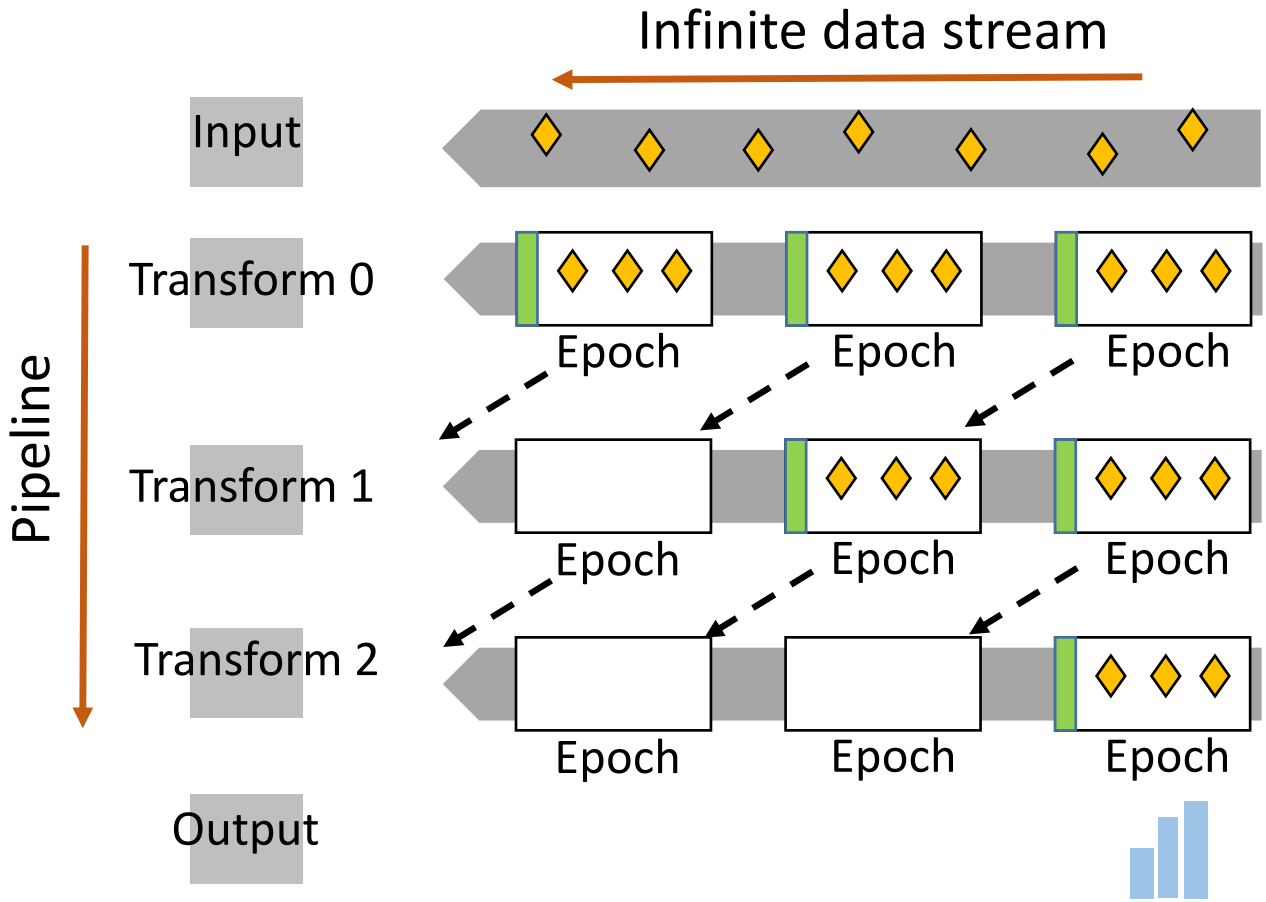
Pipeline a dataflow graph of transforms

Transform a computation that consumes and produces streams



A Simple WordCount Pipeline

Streaming Pipeline



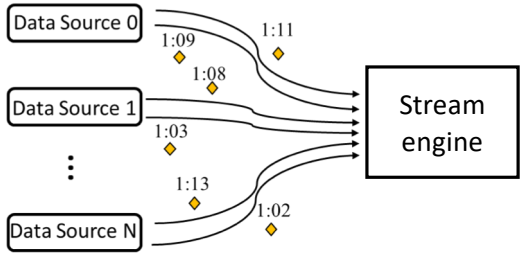
Why is it hard?

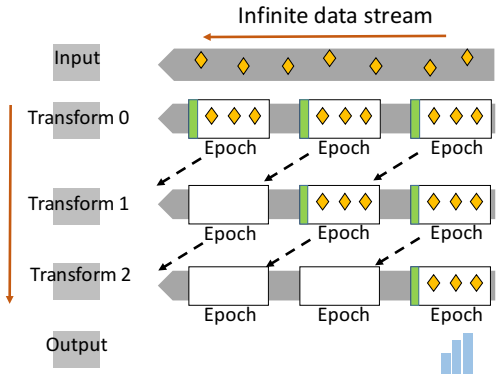
High Performance on Multicore

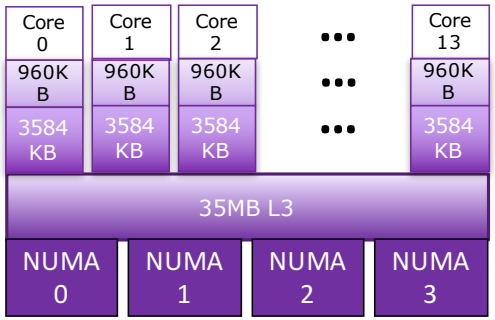
- Data parallelism (records within windows)
- Pipeline parallelism (records across pipelines)
- Memory locality (NUMA)

Records arrive out-of-order

- Records from *same data source* may arrive out-of-order
  - Different network paths
  - e.g. 1:09 and 1:11 from Data Source 0
- Records from *different data sources* may arrive out-of-order
  - Different network paths or Non-synchronized clocks
  - E.g. 1:11 from Data Source 0 and 1:08 from Data Source 1
- Challenge parallel execution on multicore
  - Cannot start processing a window before it closes
  - Have to wait for window completion





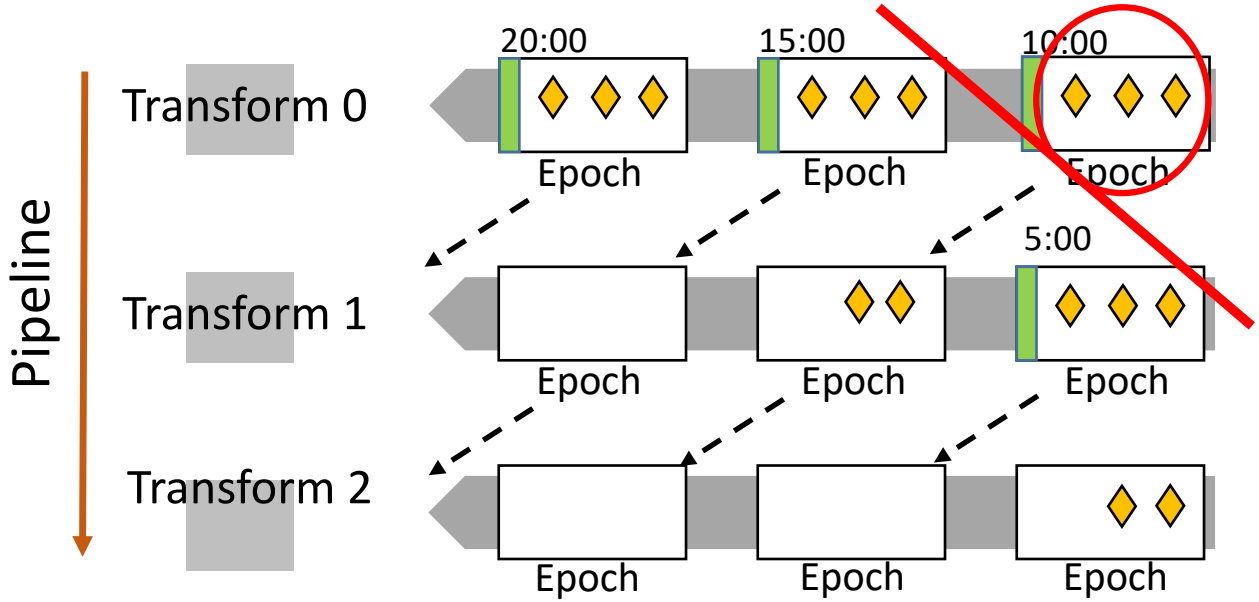


Intel Xeon E7-4830 v4

Prior work

Out-of-order processing within epochs

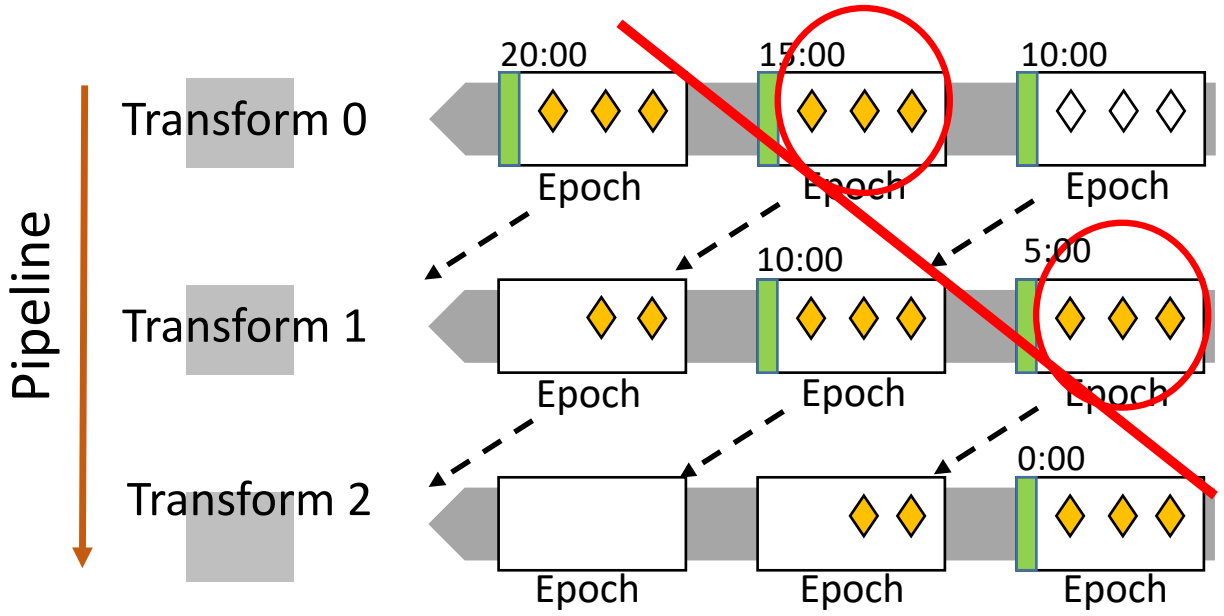
Processes only one epoch in each transform at a time



Prior work

Out-of-order processing within epochs

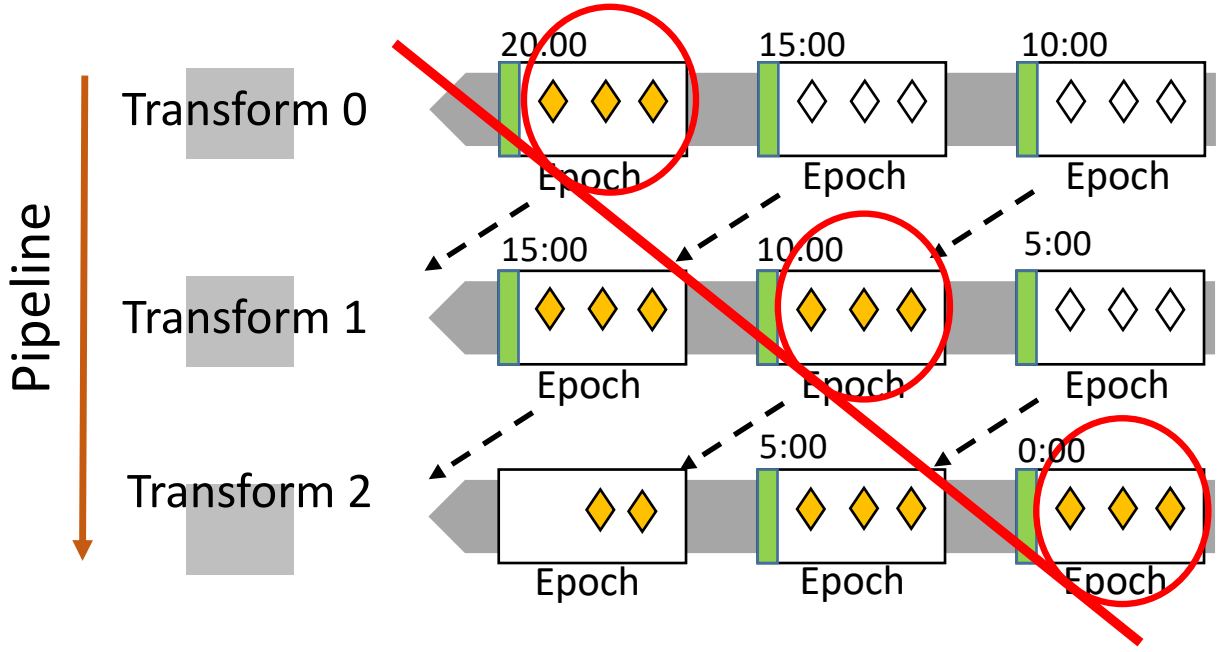
Processes only one epoch in each transform at a time



Prior work

Out-of-order processing within epochs

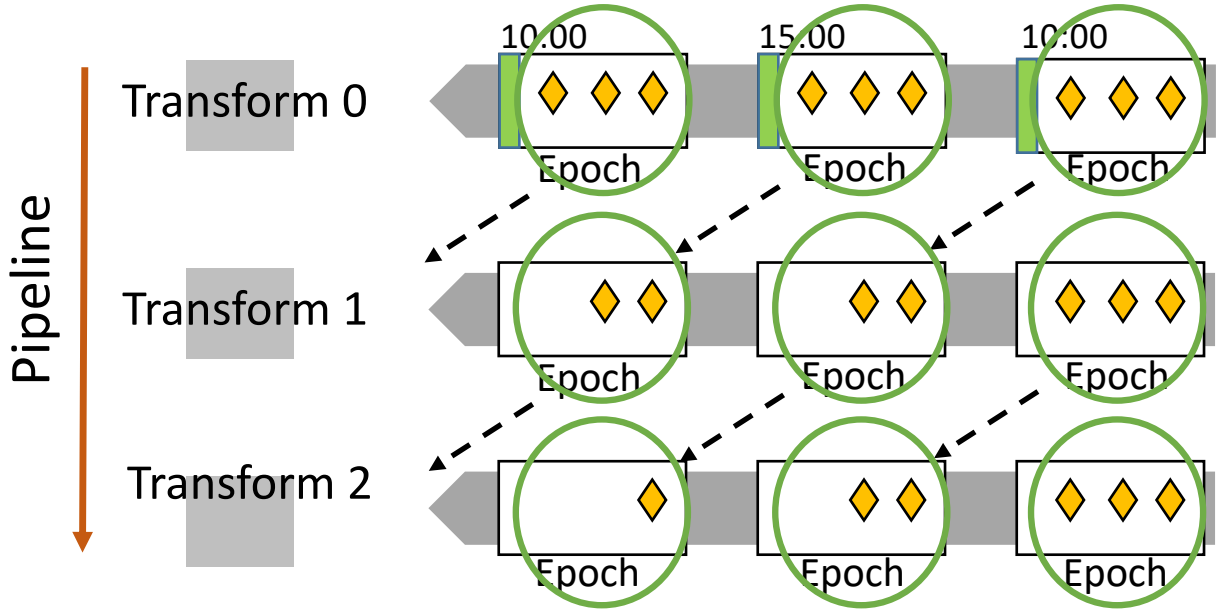
Processes only one epoch in each transform at a time



StreamBox insight

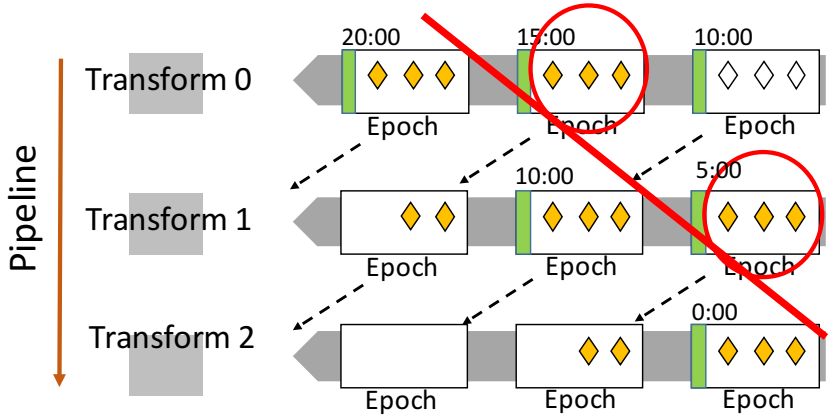
Out-of-order processing across epochs

Process all epochs in all transforms in parallel

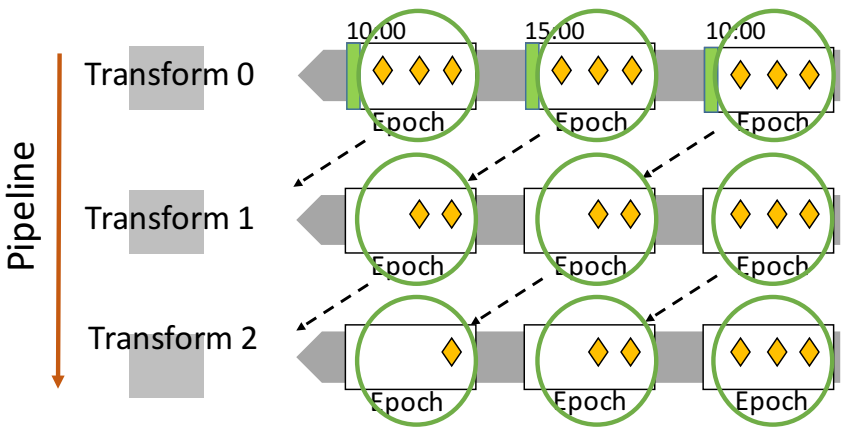


Prior work vs. StreamBox

Processes only one epoch in each transform at a time



Process all epochs in all transforms in parallel



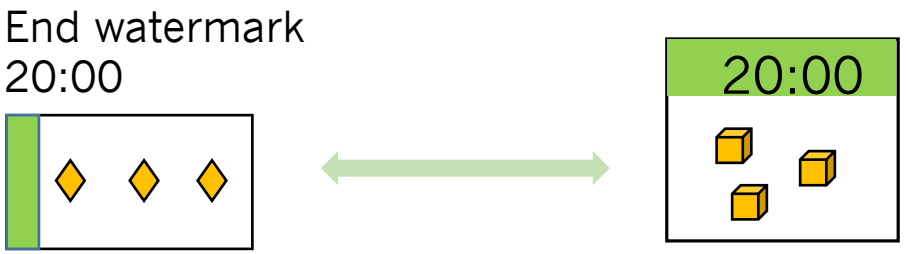
StreamBox: High pipeline and data parallel processing system

Our Key design: **Cascading containers**

Keep track of data dependencies for executing out-of-order records in parallel

Each cascading container

- Corresponds to an **epoch**
- Tracks an epoch state and the relationship between records and the watermark
- Orchestrates worker threads to consume watermarks and records



An epoch/window **A container**

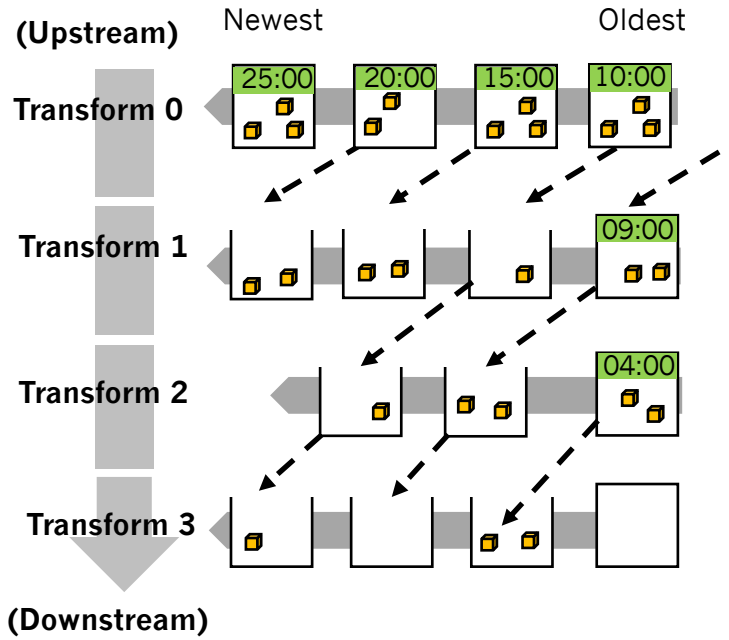
Parallel execution of out-of-order data based on cascading containers

A pipeline: multiple transforms

- Containers form a network
- Records/watermarks flow through the links

High parallel pipeline

- Guarantees watermark semantic
- Avoids stalling pipeline (for throughput)
- Avoids relaxing watermark (for latency)



Other key optimizations

Organizing records into bundles

- Minimize synchronization

Multi-input transforms

- Defer container ordering in downstream

Pipeline scheduling

- Prioritize externalization to minimize latency

Pipeline state management

- Target NUMA-awareness and coarse-grained allocation

StreamBox implementation

Built from scratch in 22K SLoC of C++11

- Supported transforms: Windowing, GroupBy, Aggregation, Mapper, Reducer, Temporal Join, Grep…
- Source code @ http://xsel.rocks/p/streambox

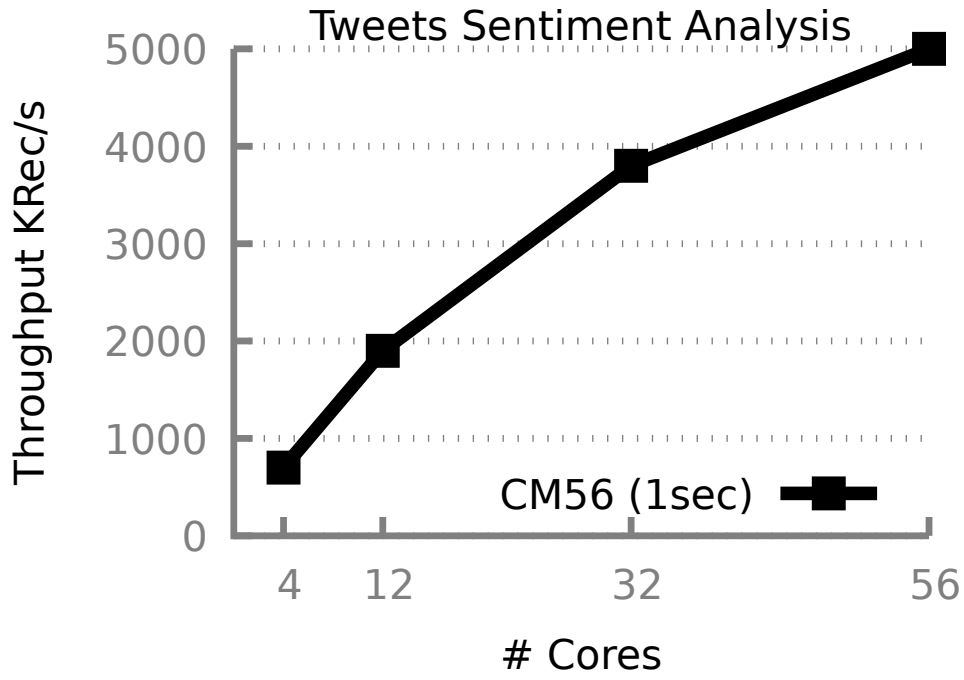
C++ libraries

- Intel TBB, Facebook folly, jemalloc, boost…

Concurrent hash tables

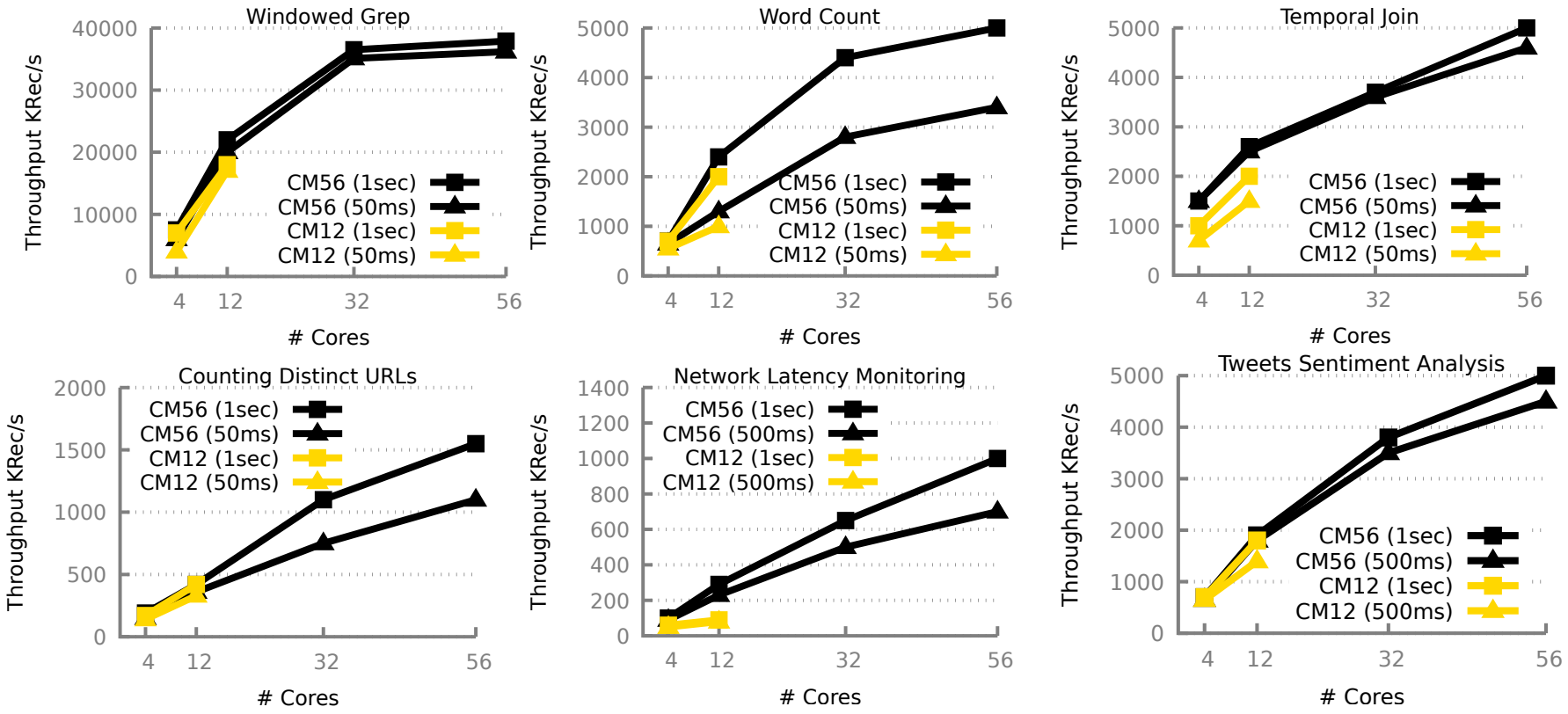
- Wrapped TBB's concurrent hash map

Good throughput and scalability



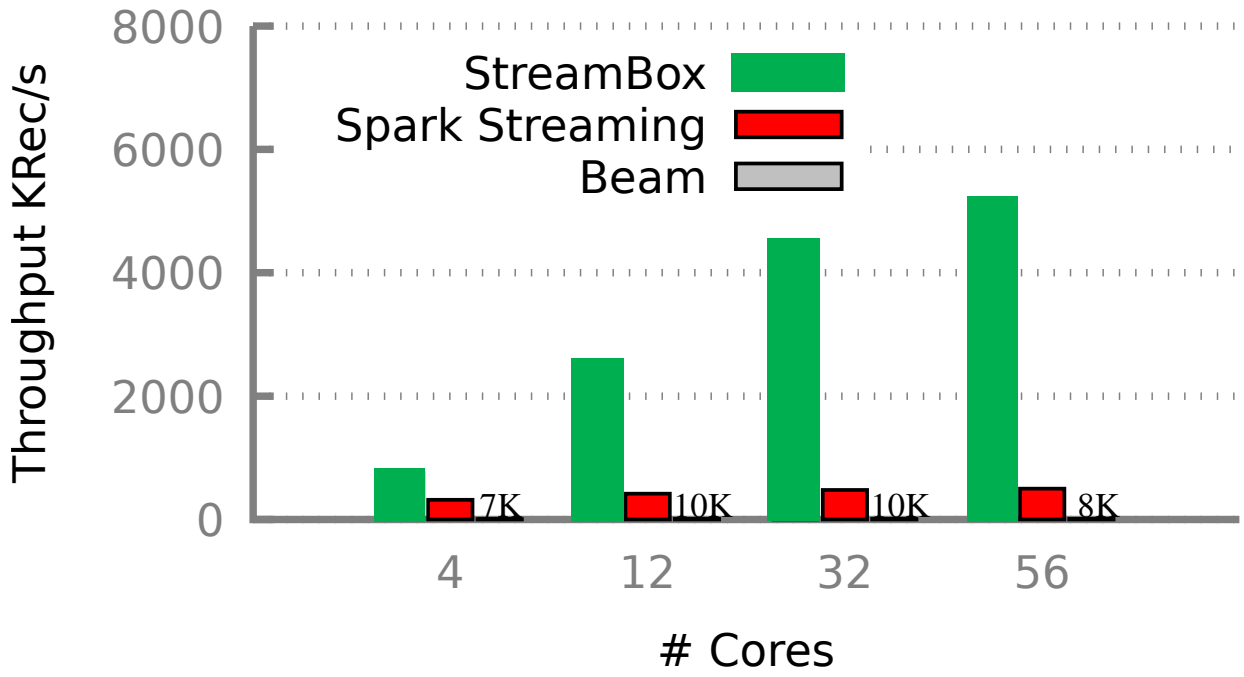
CM12: 12 cores, 256 GB DRAM. CM56: 56 cores, 256 GB DRAM

Good throughput and scalability



CM12: 12 cores, 256 GB DRAM. CM56: 56 cores, 256 GB DRAM

StreamBox vs. existing stream engines



Spark: v2.1.0
Beam: v0.5.0

StreamBox achieves significantly better throughput and scalability

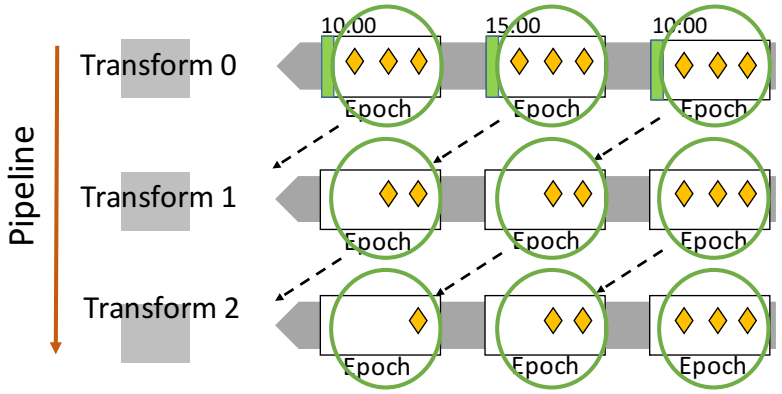
CM12: 12 cores, 256 GB DRAM. CM56: 56 cores, 256 GB DRAM

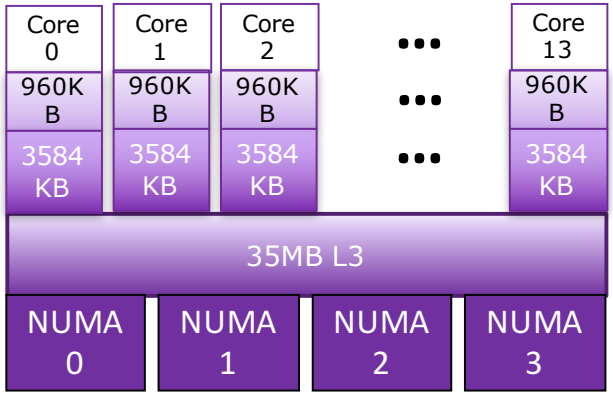
Summary: StreamBox on Multicore

Processes any records in any epochs in parallel by using all CPU cores

Achieves high throughput with low latency

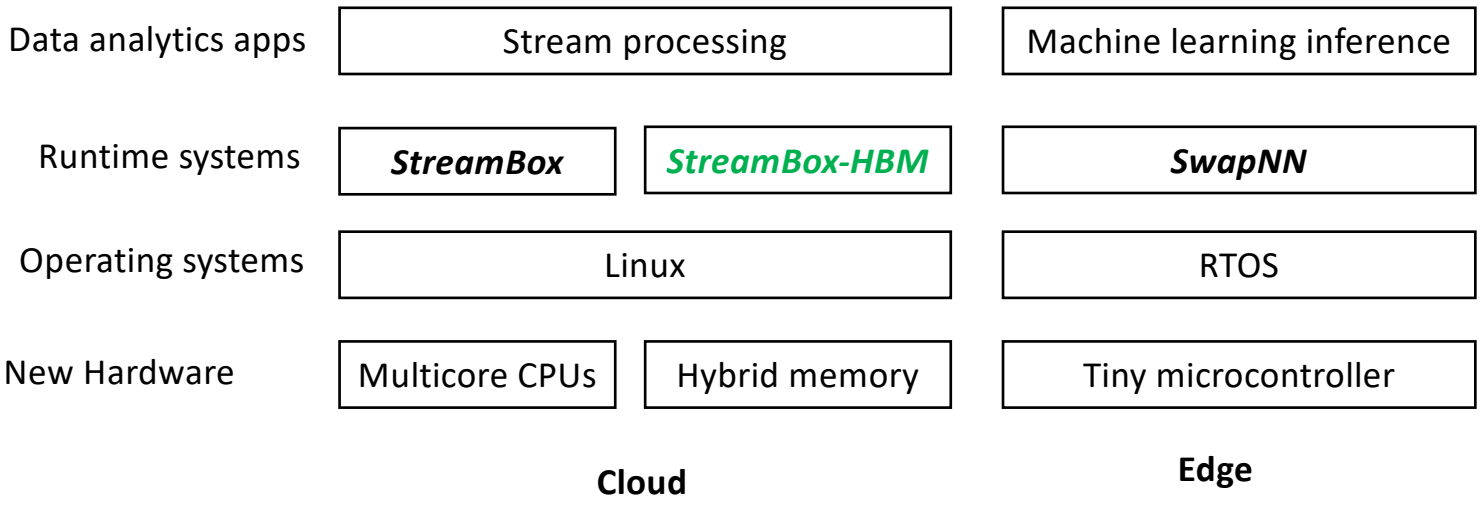
- Millions records per second throughput, on a par with distributed engines on a cluster with a 100-200 CPU cores
- Tens of milliseconds latency, 20x shorter than other large-scale engines



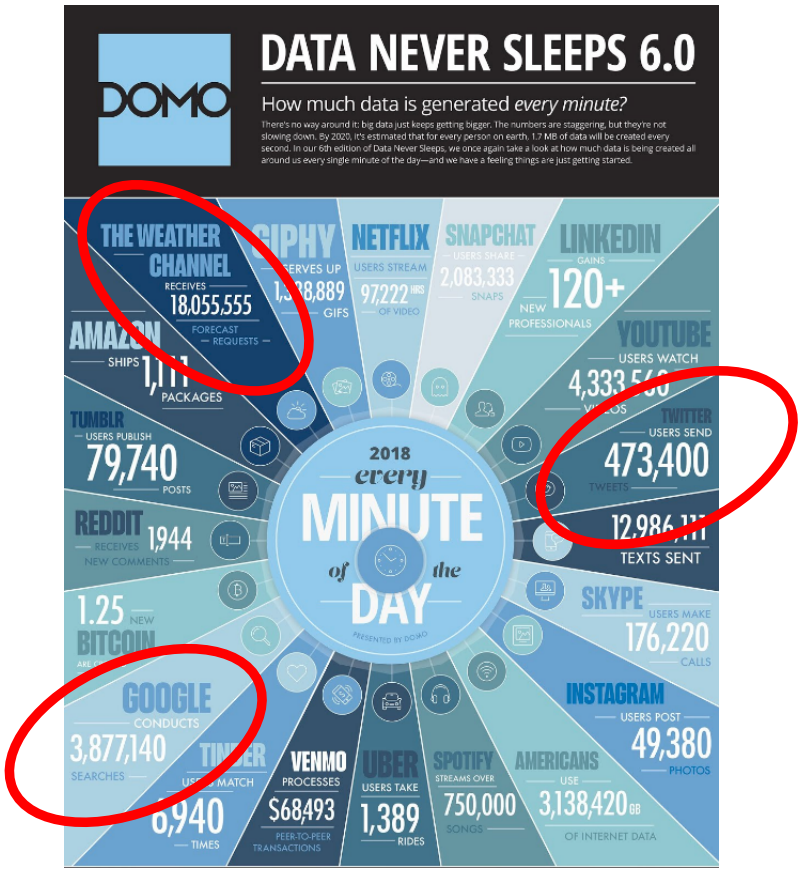


This Thesis: StreamBox-HBM

- We exploit hybrid memories to balance bandwidth and latency, achieving memory scalability and highly efficient performance
- Key results: outperform existing engines by an order of magnitude



Timely processing of streaming data



On 100+ GB memory

High Throughput & Low Latency!

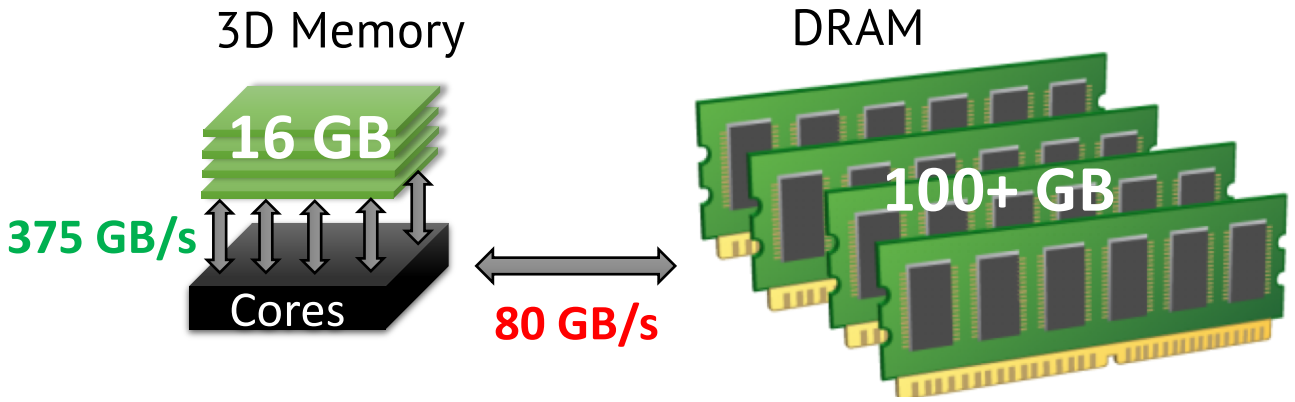
Hybrid Memory: 3D Memory + DRAM

DRAM

- Larger capacity, but lower bandwidth

3D Memory

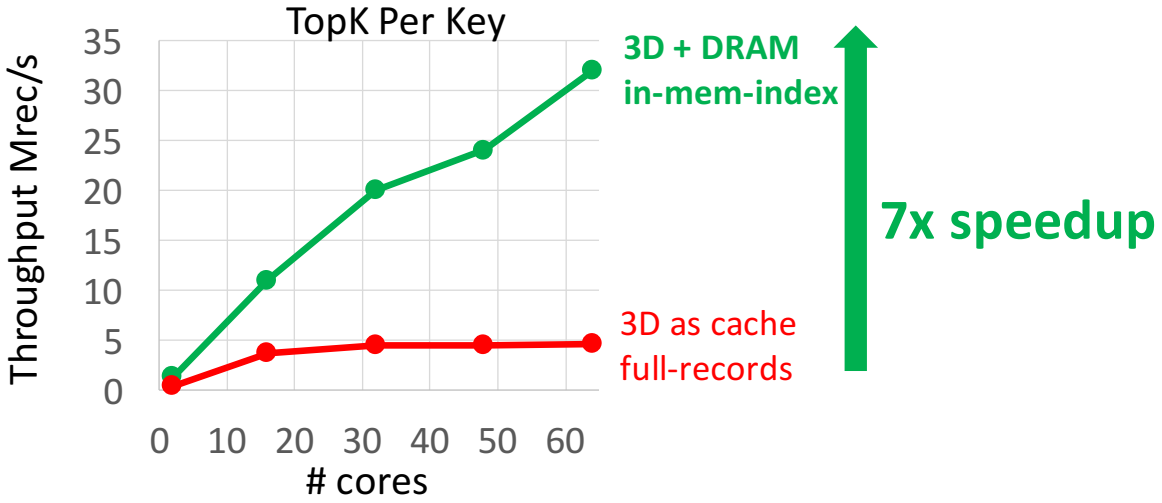
- Higher bandwidth, but smaller capacity
- NO latency benefit (Unlike cache: SRAM+DRAM)
- Same as DRAM without high parallelism or sequential access
- As cache of DRAM? → Poor performance…



Can hybrid mem speed up stream analytics?

Yes! StreamBox-HBM

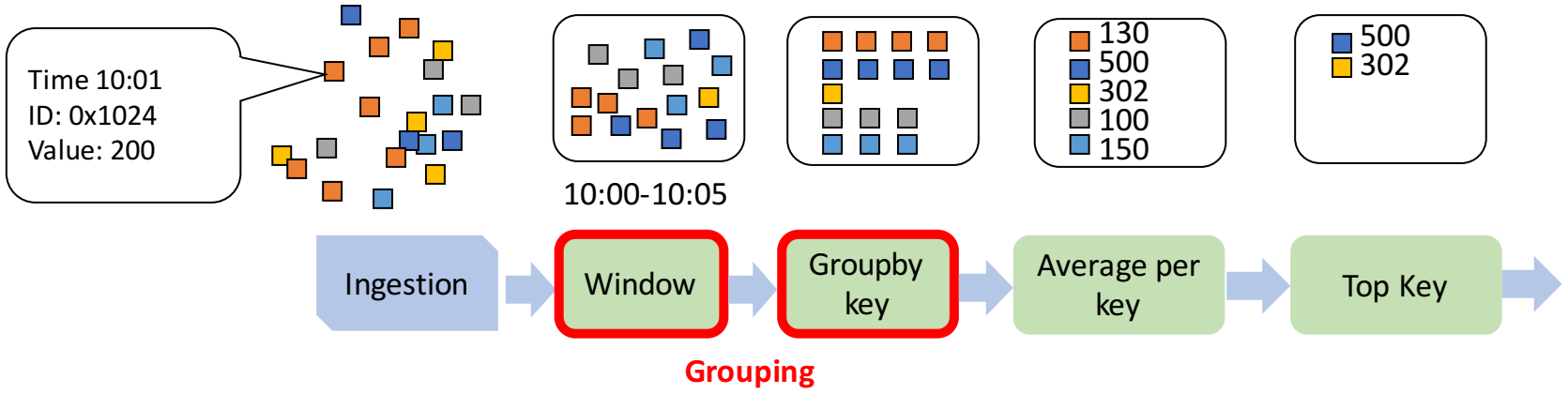
- The first stream engine optimized for 3D memory + DRAM on real hardware
- Achieves the best reported throughput on single node (win-avg:110MRec/s)
- Speeds up stream analytics by 7x



Challenge 1: Hash Grouping performs poorly on 3D memory

Data Grouping

- A set of **very common** and **expensive** operators that reorganize records
- Hash with random access in existing engines → Performs poorly on 3D memory…



Solution 1: Parallel Sort for Grouping

Known duals of Grouping: Hash vs. Sort

- DRAM: Hash is the best [VLDB'09, VLDB'13, SIGMOD'15]
- Contribution: 3D memory reverses the debate. Sort outperforms Hash.

Sort is **worse** than Hash on algorithmic complexity

- O(NlogN) vs. O(N)

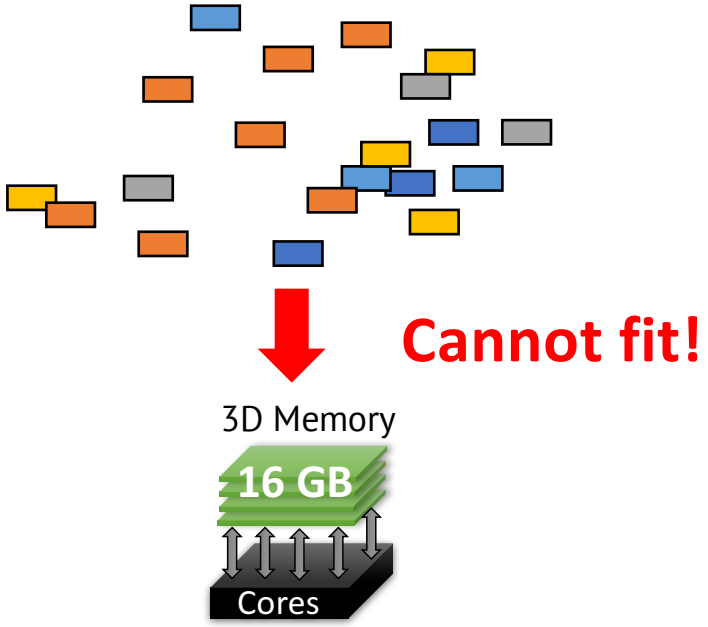
Yet, Sort **outperforms** Hash after we exploit all:

- Abundant memory bandwidth
- High task parallelism
- Wide SIMD (avx512)

[VLDB'09] Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. [VLDB'13] Multi-core, main-memory joins: Sort vs. hash revisited
[SIGMOD'15] Rethinking simd vectorization for in-memory databases

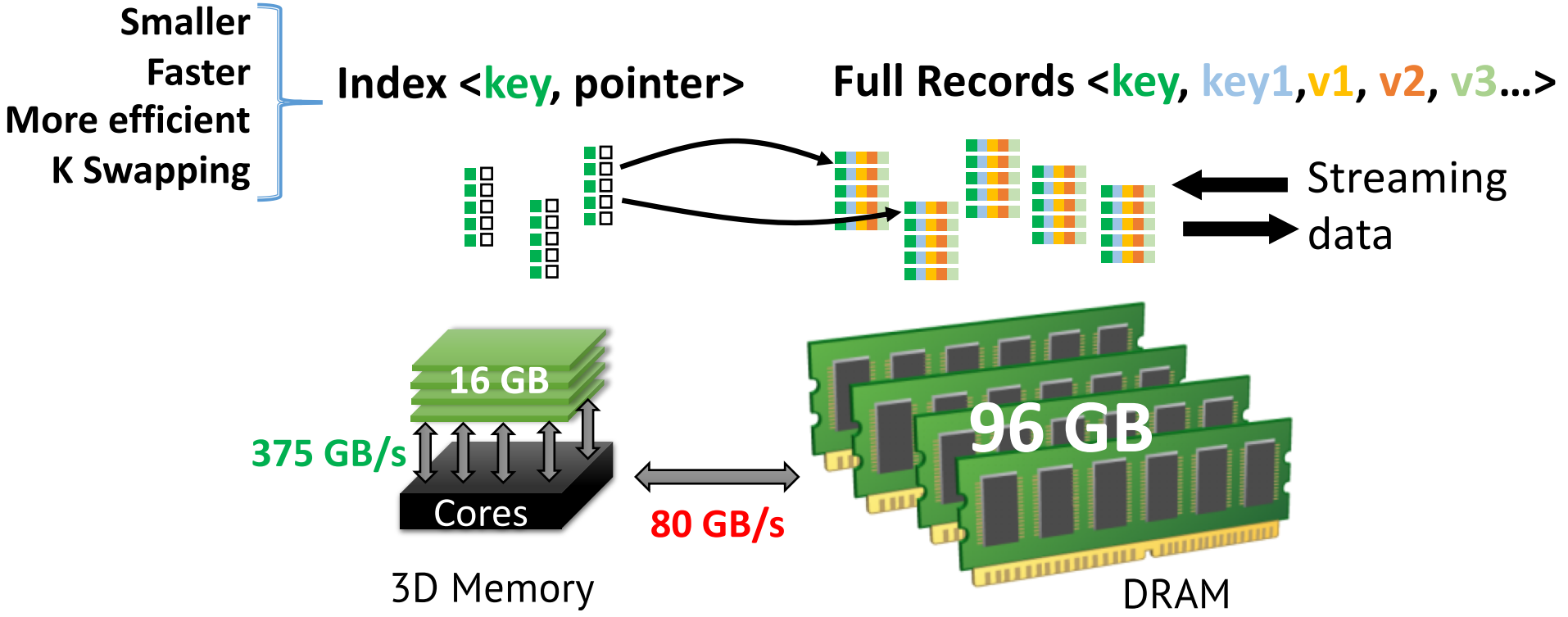
Challenge 2: 3D memory is capacity limited

- Streaming data (100+ GB)
  - High data volume
  - Input/intermediate data/output
- 3D Memory (~ 16 GB)
  - Capacity limited
  - Due to power/heat



- 3D memory is NOT large enough to hold all streaming data….

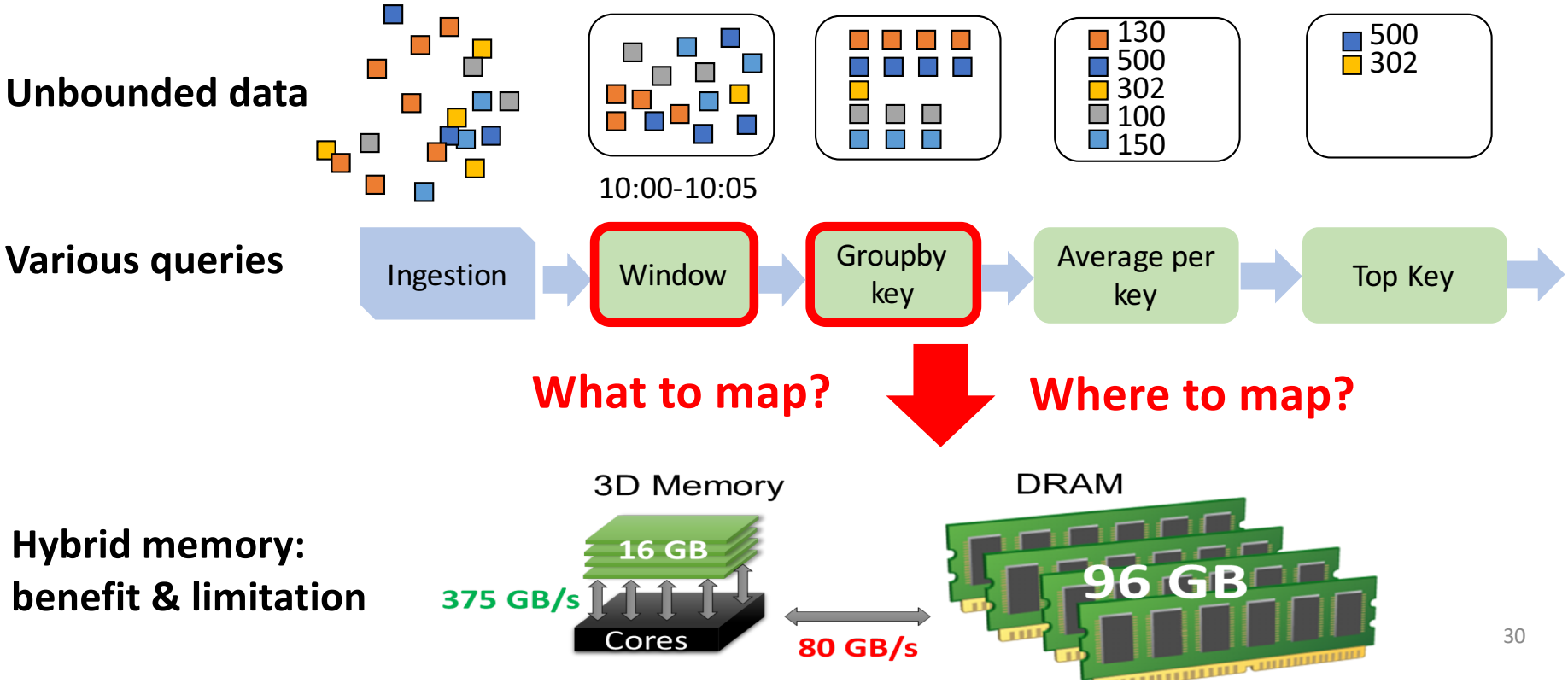
Solution 2: Only use 3D memory for in-memory index



Minimize the use of precious 3D mem's capacity while exploit high bandwidth

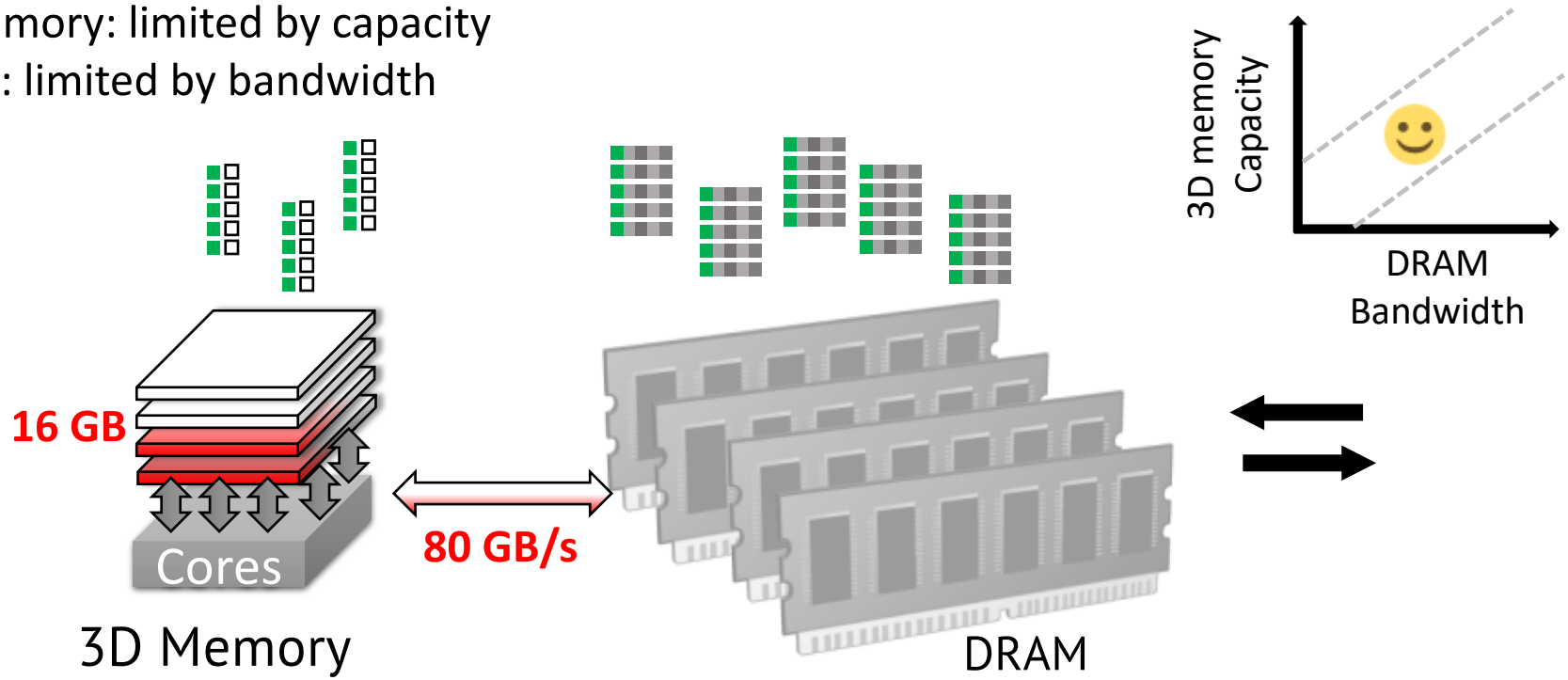
Challenge 3: managing two types of memory

- How to **dynamically** map data/operators to two types of memory?



Solution 3: balance two limited resources

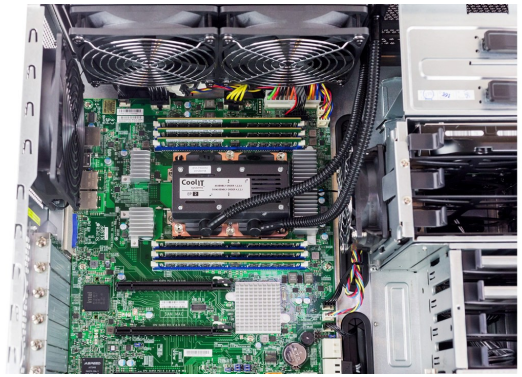
- 3D memory: limited by capacity
- DRAM: limited by bandwidth



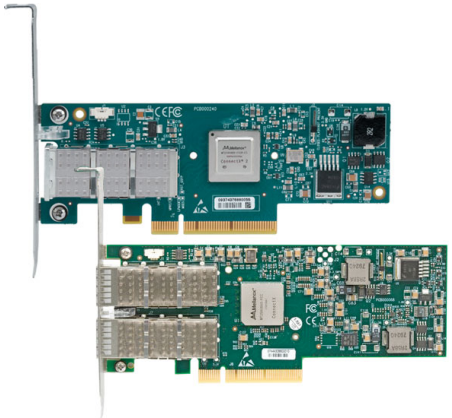
StreamBox-HBM Implementation

- Based on our prior work StreamBox
- Implement on real hardware (Intel KNL) with RDMA network
  - 61K lines of C++11, of which 38K lines are new
  - Open source: http://xsel.rocks/p/streambox

16GB 3D memory
96GB DRAM
64 cores @1.3GHz



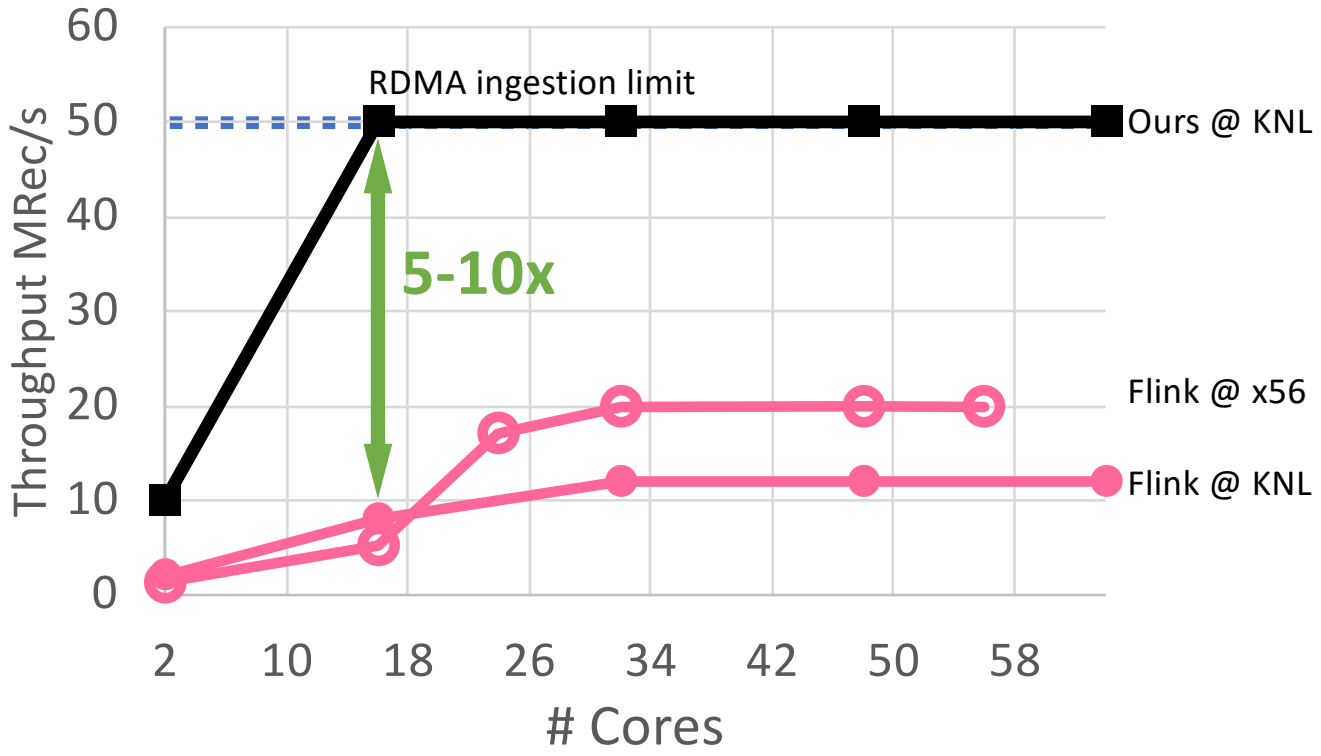
Ninja Developer Platform (KNL)



40Gb/s

Mellanox ConnectX-2

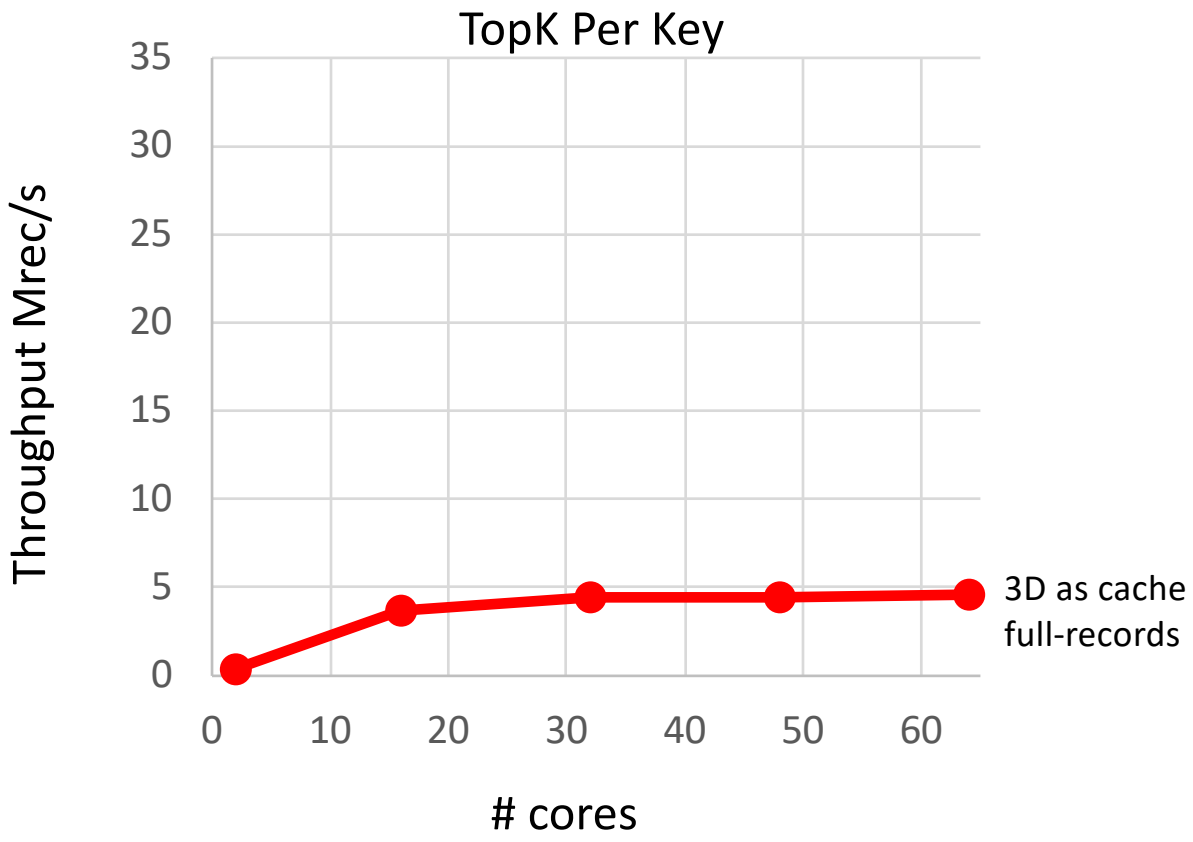
StreamBox-HBM is 10x faster than Flink



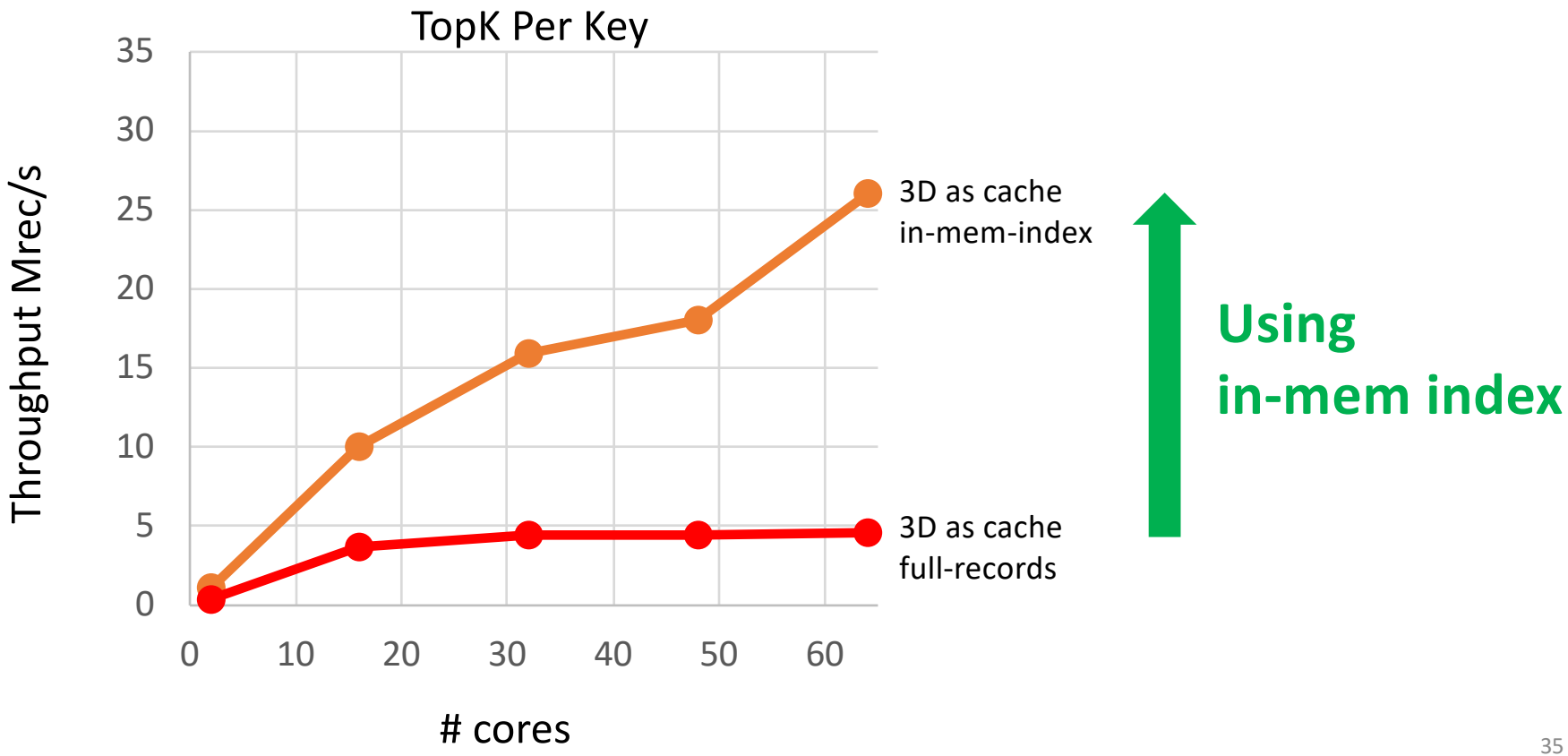
KNL: Intel Xeon Phi Knights Landing w/ HBM. 64 cores@1.3GHz. $5,000
x56: Intel Xeon E7-4830v4. 4x14 cores @2.0GHz. 256GB. $23,000

Benchmark: Yahoo Stream Benchmark.
Output delay: 1 second

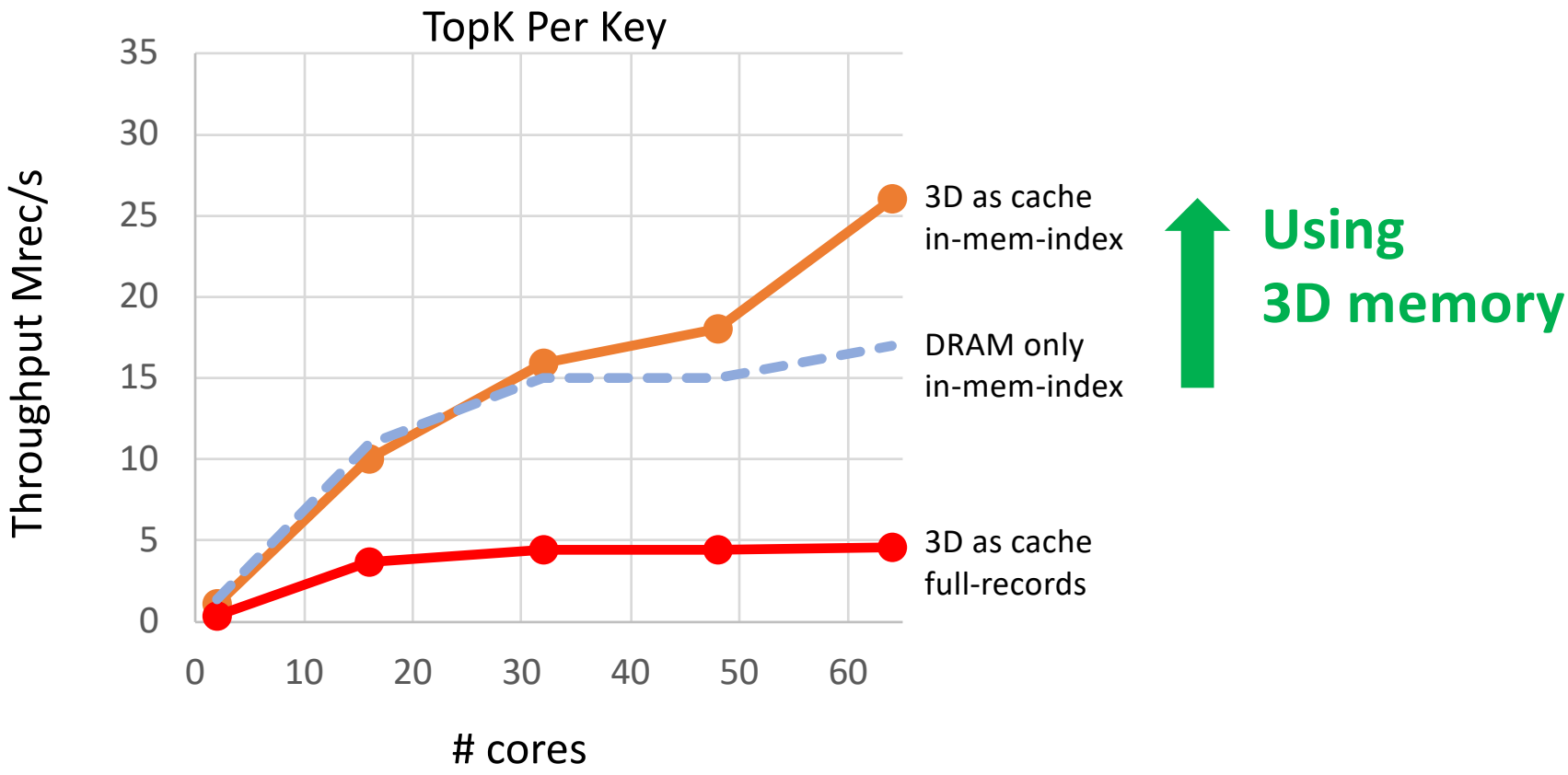
Poor performance without any key designs



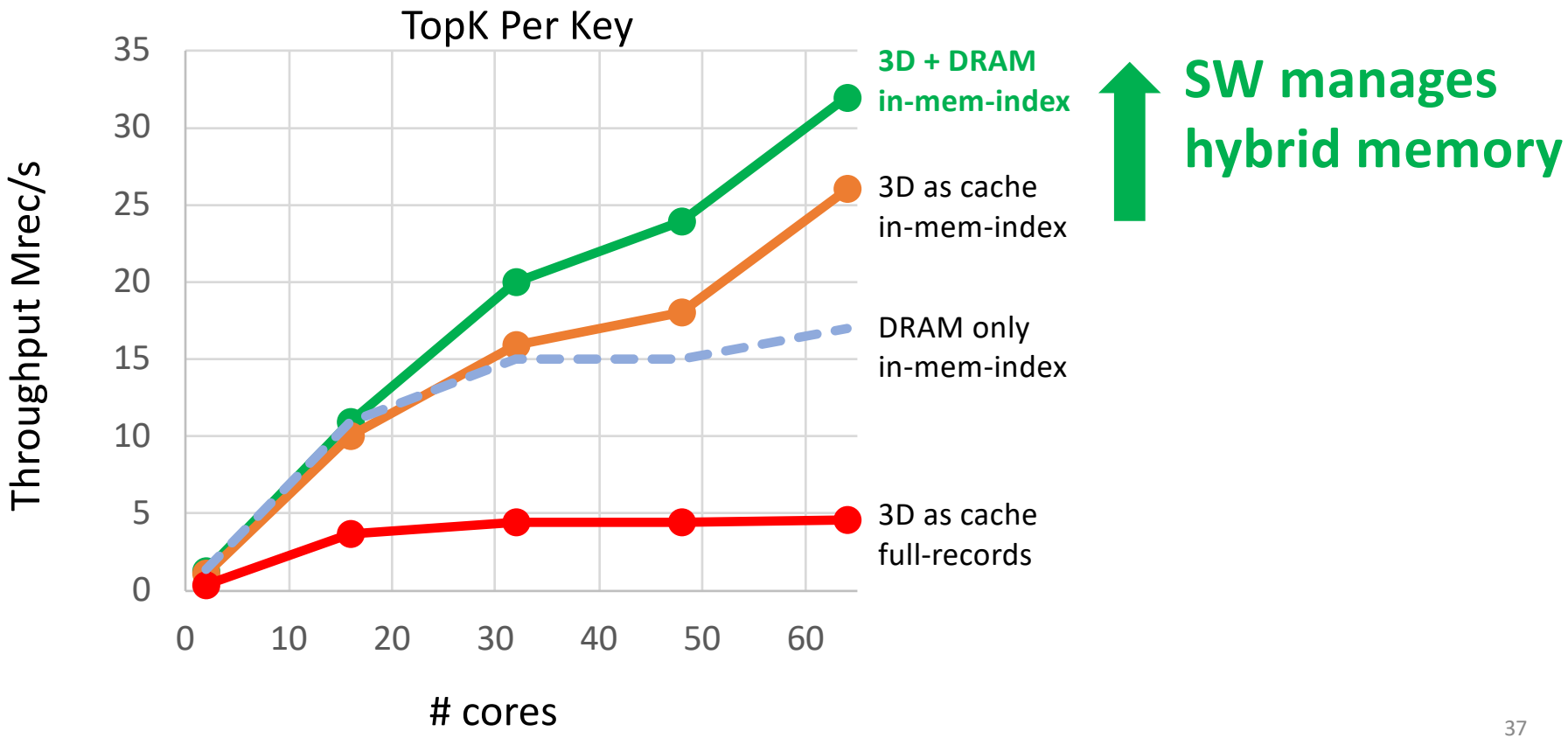
In-mem-index performs better than full-record



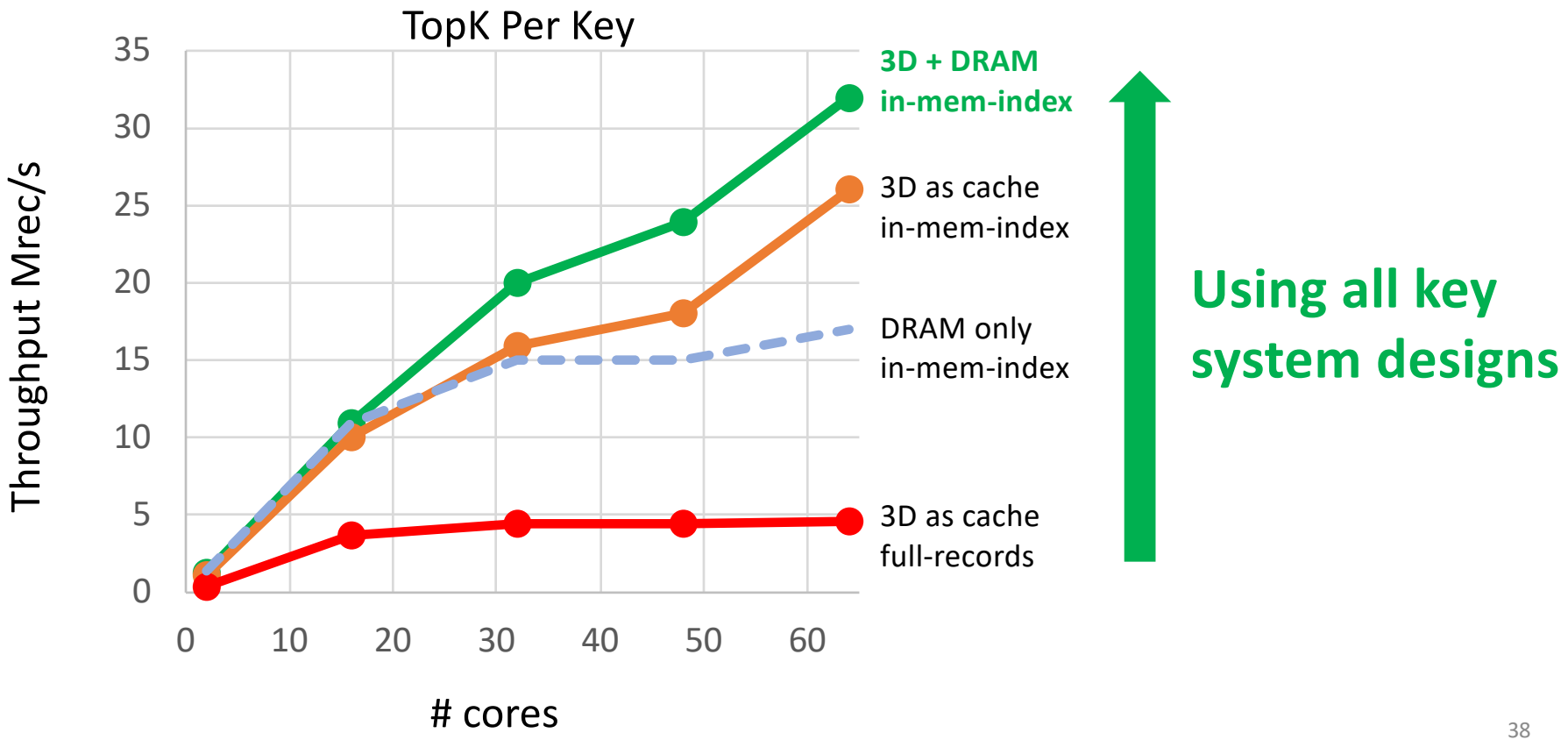
3D memory boosts performance



SW better manages hybrid memory than HW



Performance improve with all system designs



StreamBox-HBM

The first stream engine optimized for 3D Memory + DRAM on real hardware

Achieved the best reported performance on single machine

1. Grouping with Sort	2. In-memory index in 3D Memory	3. Mng hybrid mem

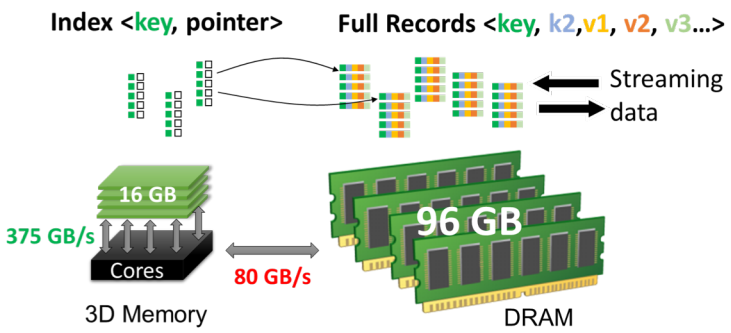
Hash → Sort

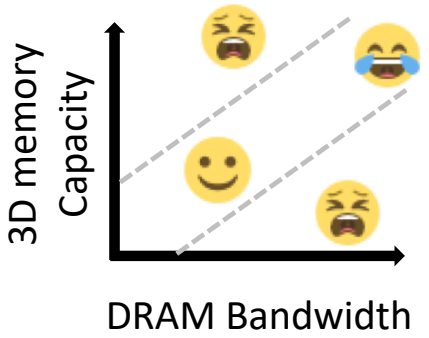
Abundant memory

High parallelism

Wide SIMD (avx512)

Sequential access

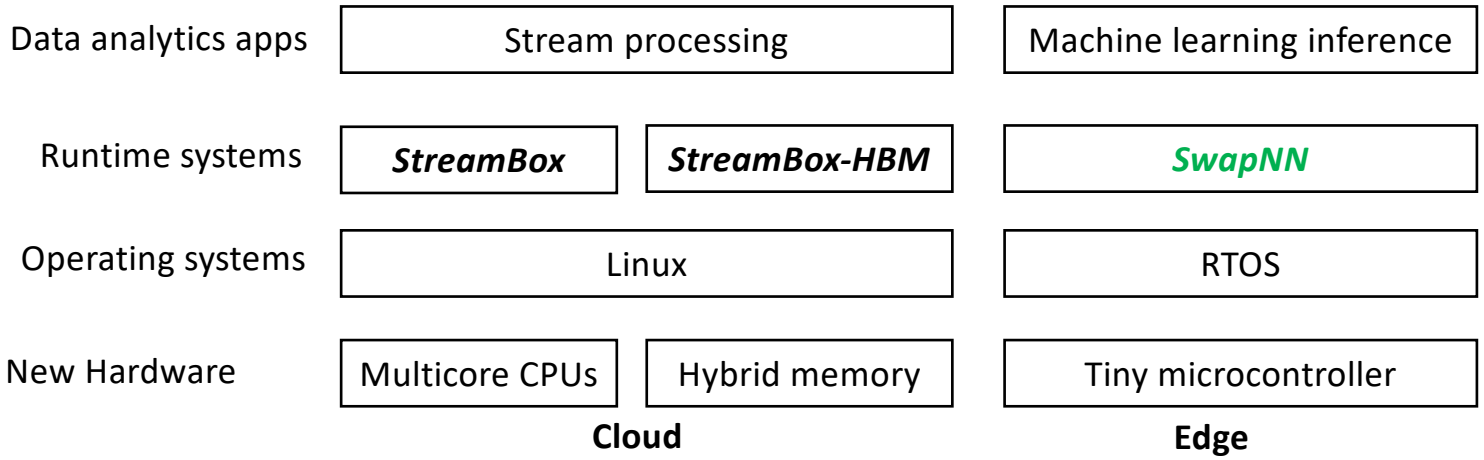




Exploit high bandwidth	Minimize use of capacity	Balance limited resources

This Thesis: SwapNN

- Goal: enable large neural networks (NNs) on tiny microcontrollers(MCUs) without losing accuracy
- A system solution for MCUs to execute NNs out of core: dynamically swapping NN data chunks between an MCU's tiny SRAM and its large, low-cost external flash.
- A study: showing that none of execution slowdown, storage wear out, energy consumption, or data security is a showstopper.
- Insights: MCUs can play a much greater role in edge intelligence



SwapNN: Out-of-core NNs on MCUs

- Background and motivation
- NN study
- Scheduler design for executing IO/compute in parallel
- Findings
  - Tradeoffs in swapping
  - Impact on slowdown: throughput and latency
  - Impact on flash durability
  - Impact on energy consumption
  - Impact on data security

Why NNs on MCUs?

- Billions of MCU devices in the world today [1]
  - 250+ billions as of now
  - 2018 alone: 28 billions sold
  - Annual shipment is keeping increasing: 38 billion by 2023

- The whole world will get a lot smarter if these MCU devices can run NNs!
  - Smart cars/homes/buildings: self-driving cars, TVs, speakers, cameras, etc.
  - Smart healthcare: monitor/detect disease based on heart rate, blood pressure, etc.
  - Smart factory: e.g., monitor/detect equipment failures
  - Smart agriculture: e.g., monitor/detect crop diseases

[1] https://venturebeat.com/2020/01/11/why-tinyml-is-a-giant-opportunity/

Why running NNs on-device locally?

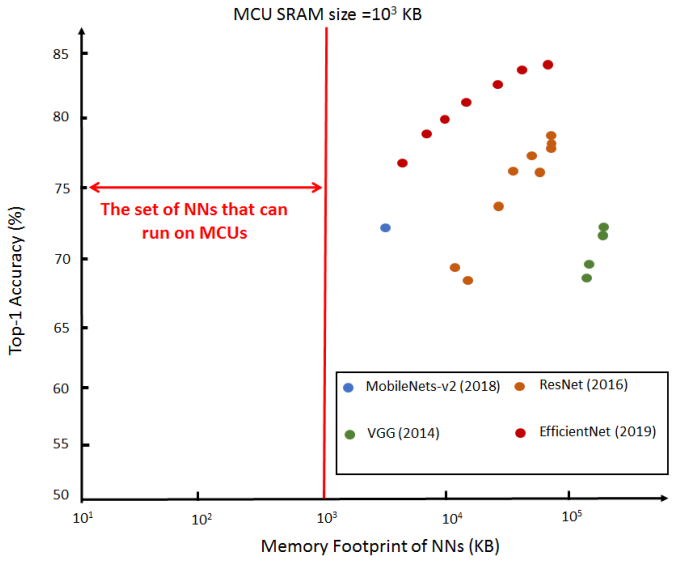
Compared to sending data to Cloud for NN inference, running NNs on MCU devices locally offers the following key advantages:

- Preserving data privacy
  - E.g., smart cameras/speakers at home carry sensitive data, sending such data to cloud incurs privacy issues
- Tolerating poor network
  - E.g., On-device NN inference can still function even without network connection

Key challenge: MCU has limited memory

- 100-1000x gap: NN's memory footprint vs. MCU's SRAM size
  - NN's memory footprint: tens to hundreds MB, e.g., VGG 100 MB
  - MCU's SRAM: few hundreds KB, e.g., 512 KB

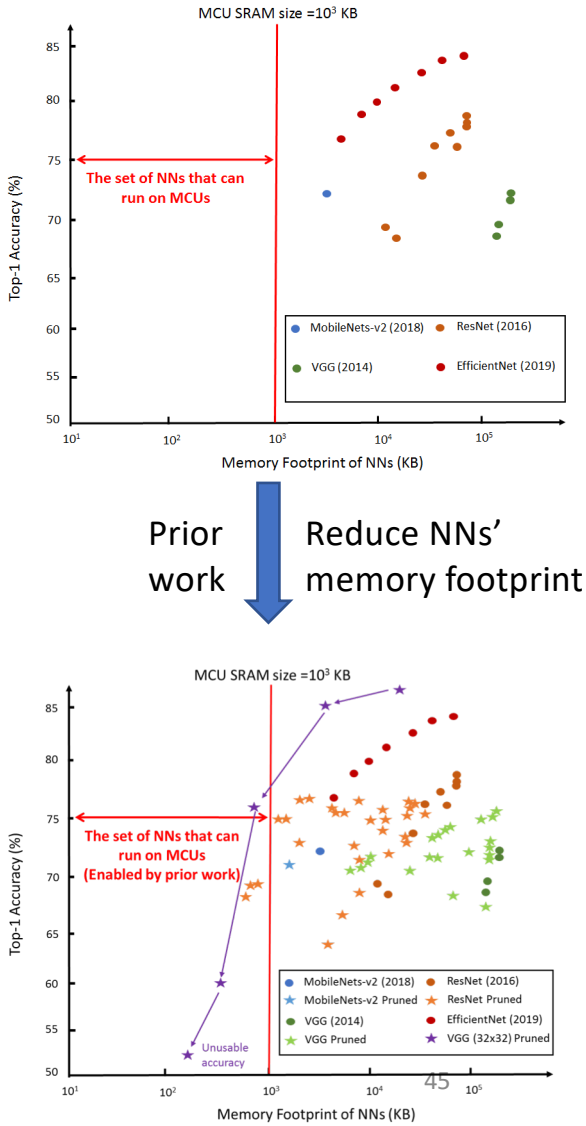
- Most NNs cannot run on MCUs due to limited memory



Prior work: algorithm-level solution

- Techniques to reduce NN memory footprint
  - Model compression (e.g., pruning)
  - Parameter quantization (e.g., float vs. integer)
  - Designing tiny NNs from scratch (e.g. binary NN)

- Key shortcomings:
  - Losing NN accuracy
  - NNs still cannot fit into MCU' SRAM after applying these techniques



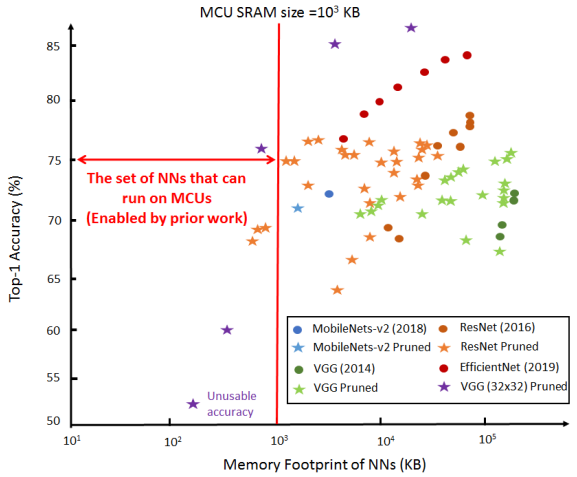
Our system-level solution: out-of-core NNs on MCUs

- Enabling large NNs on MCUs by dynamically swapping NN data chunks between an MCU's tiny SRAM and its large, low-cost external flash without losing NN accuracy
  - Why external Flash (SD card)? -> Large capacity + Low cost

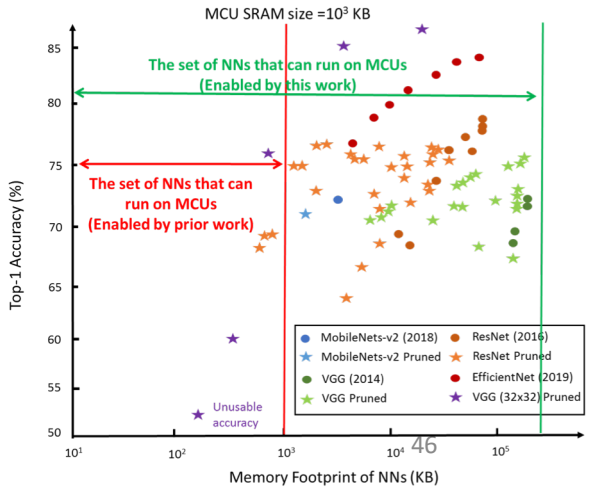
- However, swapping to SD card raises multiple concerns:
  - Loss of SD card durability?
  - Execution slowdown due to IO operations?
  - Energy increase?
  - Safety/security of out-of-core NN data?

- Our goal in this work: aims to address these concerns

Prior work



Our work



SwapNN

- Background and motivation
- NN study
- Scheduler design for executing IO/compute in parallel
- Findings
  - Tradeoffs in swapping
  - Impact on slowdown: throughput and latency
  - Impact on flash durability
  - Impact on energy consumption
  - Impact on data security

NN study: a taxonomy of NN layers

Classifying NN layers based on arithmetic intensity (ratio of computation and data move) to understand compute/IO behaviors

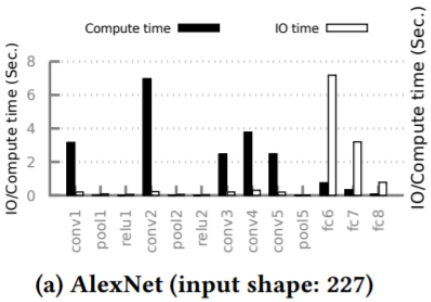
- **Compute-bound** layers (majority, N >> 1)
  - E.g., convolutional layers
  - Computation delay overshadows the IO delay
- **IO-bound** layers (minority, N < 1)
  - E.g., fully connected layers
  - Computation delay cannot overshadow IO delay, but such layers are minorities (2/12 in VGG16)
- Other insignificant overhead (minority)
  - E.g., Relu, Pooling
  - Low complexity. Tiny fraction of data to move/compute (0.3-0.9%)

Layer	Compute (MOps)	IO traffic (MB)	*N* on typical MCUs
block1_conv2	1849.69	6.46	5.96 -- 178.97
block1_pool	3.21	4.01	0.017 -- 0.50
block3_conv3	1849.69	2.19	17.55 -- 526.57
block4_pool	0.40	0.50	0.017 -- 0.50
block5_conv1	462.42	2.56	3.76 --112.89
fc1	102.76	102.79	0.02 -- 0.62
fc2	16.78	16.79	0.02 -- 0.62

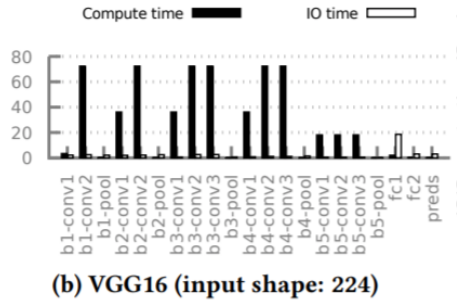
Table 1: Normalized arithmetic intensity (N) on NN layers with MCU's common speed range (64–480 MOPS [11, 13]) and IO bandwidth range (10–40 MB/s [4]). NN: VGG16

NN study: common patterns of NN layers

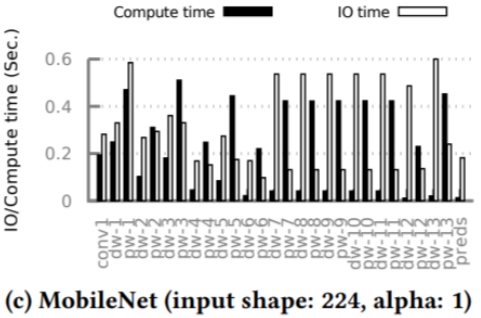
- NNs have a **mix** of **compute-bound** and **IO-bound** layers, and the number of compute-bound layers is usually bigger than other layers
- The overall NN **execution time** is dominated by the ***compute time*** **of compute-bound layer** and **the *IO time* of IO-bound layers**



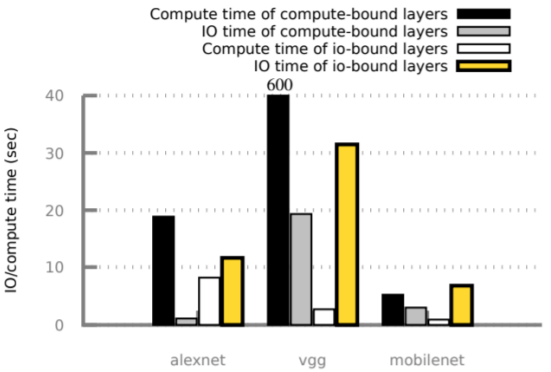
(a) AlexNet (input shape: 227)



(b) VGG16 (input shape: 224)



(c) MobileNet (input shape: 224, alpha: 1)



NN study: common patterns of NN layers

- CNNs have a mix of compute-bound and IO-bound layers, and the number of compute-bound layers is usually larger than other layers
- The overall NN execution time is dominated by the compute time of compute-bound layer and the IO time of IO-bound layers

- Insights:
  - Towards lowering the swapping overhead, we exploit the NN patterns.
  - By executing compute-bound layers and IO-bound layers in parallel, we hide the IO delays behind the compute delays.

SwapNN

- Background and motivation
- NN study
- Scheduler design for executing IO/compute in parallel
- Findings
  - Tradeoffs in swapping
  - Impact on slowdown: throughput and latency
  - Impact on flash durability
  - Impact on energy consumption
  - Impact on data security

Design: Extracting CPU/IO parallelism for hiding IO delays

- Tile parallelism (within an NN layer)
  - while computing Tile0, MCU can pre-load input for computing next tile Tile1
  - while writing back the completed Tile0 to flash, MCU can compute Tile1 simultaneously.
- Layer parallelism (within one frame)
  - MCU can execute an earlier layer's computation with a later layer's IO simultaneously
- Pipeline parallelism (across frames)
  - MCU can execute compute-bound and IO-bound layers for different frames in parallel
  - as these layers exercise complementary resources namely CPU and IO bandwidth

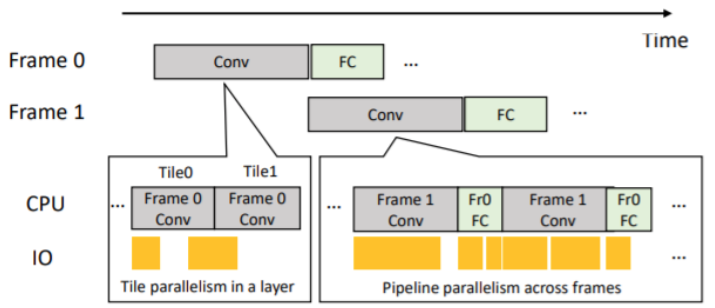


Figure 4: An example of out-of-core NN execution, showing Conv (compute-bound) and FC (IO-bound) layers.

Design: Parallel scheduler

- Scheduling goal: keep both MCU and IO busy to avoid either of them from idling
  - To hide IO with compute for low latency and high throughput
- Scheduling constraints: memory size, dependencies, and priority

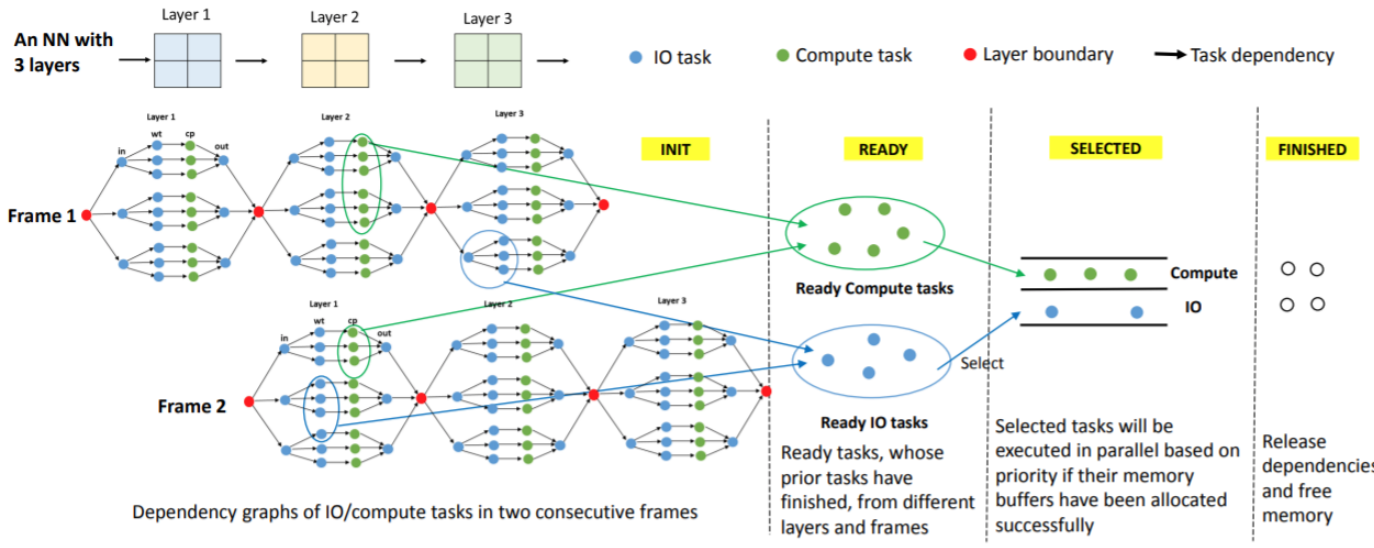


Figure 5: Overview of SwapNN: scheduling IO/compute tasks across tiles, layers, and frames in parallel according to dependencies, priorities, and memory constraints.

Experiment setup

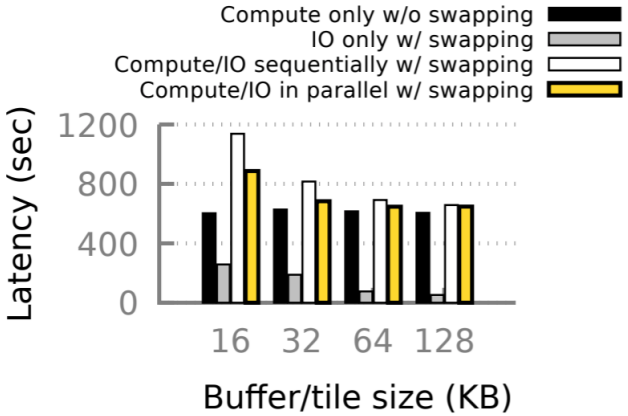
- Calculate tiling sizes for each layer based on SRAM size and NN architecture
  - Split SRAM into fixed size buffers, then calculate tile size based on buffer size
- Measure IO/compute time of each tile/task
  - measured on real MCU HW
- Find out the best parallel scheduling sequences of IO/compute tasks based on dependencies, priority, and memory constraint
  - run scheduler on a desktop without deployment

SwapNN

- Background and motivation
- NN study
- Scheduler design for executing IO/compute in parallel
- Findings
  - Impact on slowdown
  - Impact on flash durability
  - Impact on energy consumption
  - Impact on data security

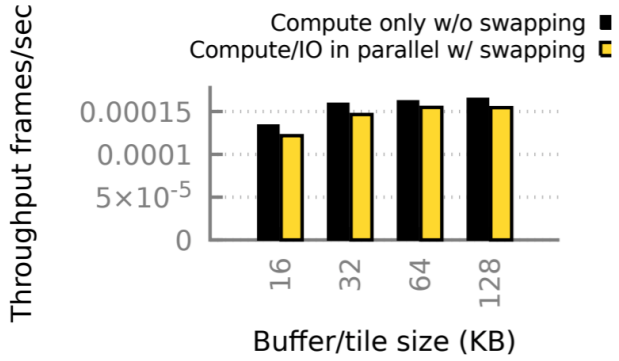
Findings: Low to modest speed overhead

- NNs with dominant compute-bound layers see negligible swapping overhead, both in per-frame delay and frame throughput.
  - VGG: #of compute-bound layer = 13, # of IO-bound layer = 2
  - Compared to in memory only, out-of-core executing VGG sees only 6.9% longer per-frame delay and only 3% lower throughput



(a) VGG, SRAM 512KB

Per-frame delay: Black vs. Yellow

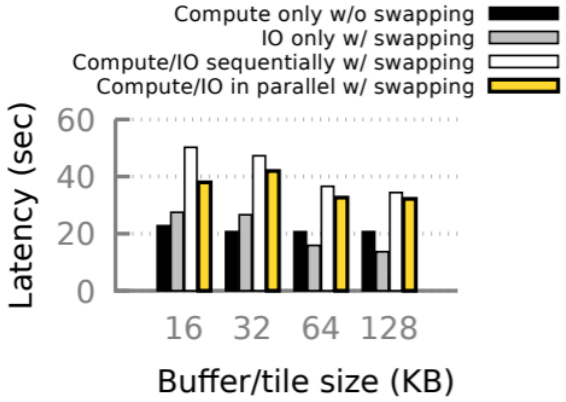


(a) VGG, SRAM 512KB

Frame throughput: Black vs. Yellow

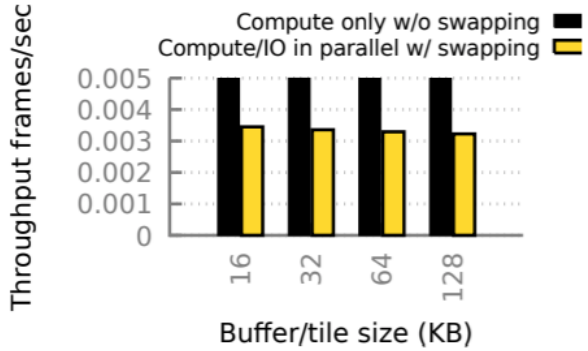
Findings: Low to modest speed overhead

- NNs with more IO-bound layers, such as AlexNet
  - see notable delay increase (50%)
  - while insignificant loss in throughput (15.7%) thanks to tile and pipeline parallelism



(e) AlexNet, SRAM 512KB

Per-frame delay: Black vs. Yellow

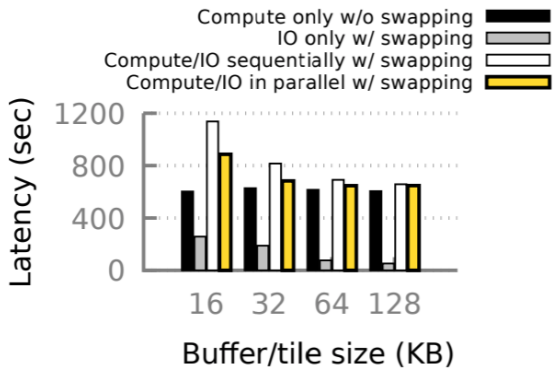


(e) AlexNet, SRAM 512KB

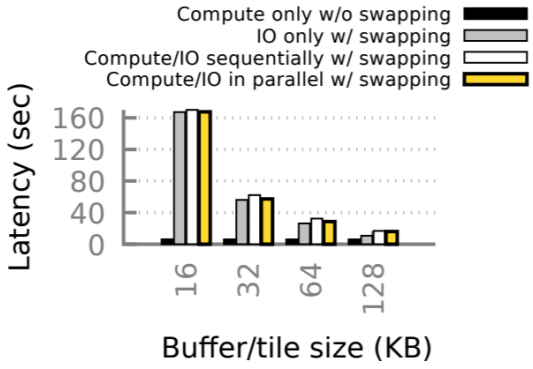
Frame throughput: Black vs. Yellow

Findings: Large tiles are crucial to low swapping overhead

- Tile size: key parameter to determine the granularity of IO/compute task
- Small tile
  - Leads to fine-grained tasks and better IO/compute parallelism (Good)
  - But increases the total amount of IO traffic and the per-byte IO delay (Bad)
- Finding
  - The cost of small tiles overshadow the benefit of parallelism on typical NNs and MCUs



(a) VGG, SRAM 512KB



(i) Mobilenet, SRAM 512KB

Findings: Low durability loss

- Even with an MCU executing NNs continuously, the write traffic due to swapping is no more than a few hundred GBs per day
  - SD card can sustain several to tens of years
  - e.g., a 64 GB SD card can sustain 7.5 years before half of its cells are worn out
- Swapping rate is throttled by computation, which limits the wear rate of SD cards
  - IO-bound layers are spaced by compute-bound layers
  - Even with continuous parallel NN execution, IO is only exercised intermittently
- Most IO traffic for swapping is read, which does not wear SD card
  - Read IO: for reading input feature map and weight parameters
  - Write IO: for writing output
  - Read IO > write IO: input feature map + weight parameters >> output feature map

Findings: Modest increase in energy consumption

- Our **worst case** estimation shows swapping increases system energy by less than 42% compared to running NNs with infinite memory (all in memory without swapping)
- Experiment setup (two threads)
  - IO thread: keep writing data to SD (with DMA support)
  - Compute thread: keep running a small NN in MCU SRAM
  - Using power meter to measure the energy of executing the NN for 1000 times
- Real energy consumption will be lower
  - IO won't be always busy

Findings: Out-of-core data can be secured with known mechanisms

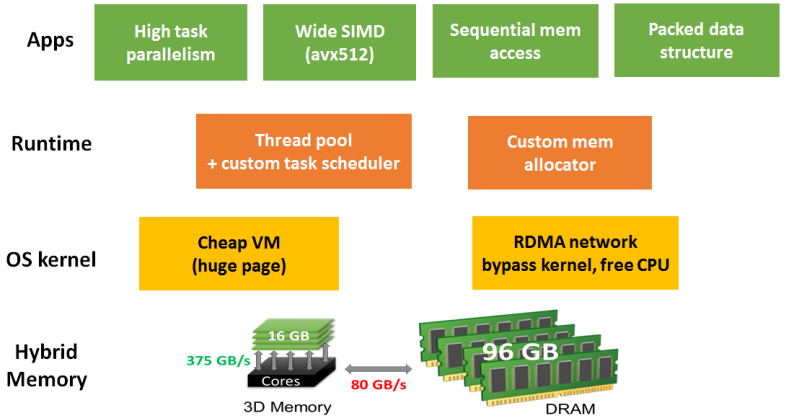
- Compared to storing NN data in on-chip SRAM, (temporarily), storing it off-chip is more vulnerable to physical attacks:
  - adversaries may learn or corrupt the data by tapping into the IO bus between MCU and the SD card, or the SD card itself
- By encrypting NN data before swapping out, MCU can ensure the data to be confidential and integral
- Specialized hardware on MCUs further reduces their overhead.
  - ASE is already common on modern MCUs. Its computation overhead is comparable to (or even less than) the least intensive NN compute.

Summary of SwapNN

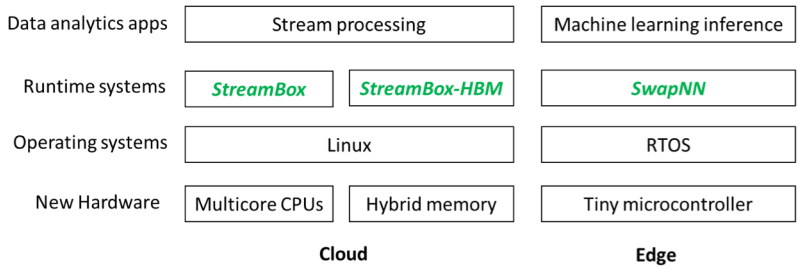
- We present the first study of applying swapping to NN on MCUs
  - Analyze the swapping-generated IO activities and their implications on performance, storage durability, energy, and data security
- We present a scheduler design that automatically schedules IO/compute tasks in parallel by exploiting common patters of NN layers
- We make a case that an MCU of less than ten dollars with hundreds of KBM SARAM can execute large NNs such as VGG, which expands the scope of edge intelligence

This Thesis: general guidance for systems designs for new applications on future hardware

- High performance requires full-stack system designs/optimizations across SW/HW
- Apps: Algorithms adapting to hardware changes
  - Sort vs. hash on HBM
- Runtime: Better managing resources than general hardware and OS
  - Managing hybrid memory: runtime vs. OS/HW
- OS: Configuring kernel parameters accordingly
  - Huge page to reduce memory map overhead
- Hardware: Choosing hardware based on applications' demands
  - Multicore/HBM for data intensive applications



Summary of the thesis



- Theme: Systems support for data analytics by exploiting modern hardware
  - Bridge the gap between data explosion and hardware evolving
- Systems support for stream processing
  - By exploiting many-core CPUs [StreamBox, USENIX ATC'17]
  - By exploiting high bandwidth hybrid memory [StreamBox-HBM, ASPLOS'19]
  - Both offer orders of magnitude performance improvements over the prior state of the art, opening up new applications with higher data processing needs
- Systems support for machine learning inference
  - Enable on-device ML inference on tiny MCUs without losing accuracy [SwapNN, arXiv'21]
  - Enable new use cases and significantly expand the scope of edge intelligence
- General guidance on system designs
  - System designs/optimizations cross hardware/software stack (app, algorithm, runtime, OS, hardware) for a wide range of new applications on future hardware platforms