SYSTEMS SUPPORT FOR DATA ANALYTICS BY EXPLOITING MODERN HARDWARE

by

Hongyu Miao

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Electrical and Computer Engineering

West Lafayette, Indiana

December 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Felix Xiaozhu Lin, Chair

Department of Computer Science, University of Virginia

Dr. Kathryn S. McKinley

Google Inc.

Dr. Mithuna S. Thottethodi

School of Electrical and Computer Engineering, Purdue University

Dr. Y. Charlie Hu

School of Electrical and Computer Engineering, Purdue University

Approved by:

Dr. Dimitrios Peroulis

School of Electrical and Computer Engineering, Purdue University

To my entire family.

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Felix Xiaozhu Lin, for his advice, help, and support. In addition to guidances in research, he also provided all resources I need to complete my PhD study, including equipments and fundings. As Felix's first PhD student, we fell into many traps and learned many lessons together in the first few years, which not only is a fortune for me but also paves the way for other/future PhD students in our group.

I would like to thank Dr. Kathryn S. McKinley for her advice, help, and support. In addition to guidance in research, her kindness, supportiveness, and responsiveness inspire me. If I had chances in the future, I would like to do the same thing to support junior people as what Kathryn did.

I would like to thank the rest of my committee, Prof. Y. Charlie Hu and Prof. Mithuna S. Thottethodi, for their feedback, questions, and discussions on my thesis work.

I would like to thank Prof. Gennady Pekhimenko and Prof. Myeongjae Jeon for their advice, help, and support through my MSR internship and follow-up collaborations.

Finally, I especially would like to thank my entire family for their long-term care, love, and support, so that I can focus on my PhD study. Without them, I could not have done it.

Contents

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A large volume of data is continuously being generated by data centers, humans, and internet of things (IoT). In order to get useful insights, such enormous data must be processed in time with high throughput, low latency, and high accuracy. To meet such performance demands, a large body of new hardware is being shipped by vendors, such as multi-core CPUs, 3D-stacked memory, embedded microcontrollers, and other accelerators.

However, traditional operating systems (OSes) and data analytics frameworks, the key layer that bridges high-level data processing applications and low-level hardware, fails to deliver these requirements due to quickly evolving new hardware and increases in explosion of data. For instance, general OSes are not aware of the unique characters and demands of data processing applications. Data analytics engines for stream processing, e.g., Apache Spark and Beam, always add more machines to deal with more data but leave every single machine underutilized without fully exploiting underlying hardware features, which leads to poor efficiency. Data analytics frameworks for machine learning inference on IoT devices cannot run neural networks that exceed SRAM size, which disqualifies many important use cases.

In order to bridge the gap between the performance demands of data analytics and the new features of emerging hardware, in this thesis we exploit runtime system designs for high-level data processing applications by exploiting low-level modern hardware features. We study two important data analytics applications, including real-time stream processing and on-device machine learning inference, on three important hardware platforms across the Cloud and the Edge, including multicore CPUs, hybrid memory system combining 3D-stacked memory and general DRAM, and embedded microcontrollers with limited resources.

In order to speed up and enable the two data analytics applications on the three hardware platforms, this thesis contributes three related research projects. In project StreamBox, we exploit the parallelism and memory hierarchy of modern multicore hardware on single machines for stream processing, achieving scalable and highly efficient performance. In project StreamBox-HBM, we exploit hybrid memories to balance bandwidth and latency, achieving memory scalability and highly efficient performance. StreamBox and StreamBox-

HBM both offer orders of magnitude performance improvements over the prior state of the art, opening up new applications with higher data processing needs. In project SwapNN, we investigate a system solution for microcontrollers (MCUs) to execute neural networks (NNs) inference out-of-core *without losing accuracy*, enabling new use cases and significantly expanding the scope of NN inference on tiny MCUs.

We report the system designs, system implementations, and experimental results. Basing on our experience in building above systems, we provide general guidances on designing runtime systems across hardware/software stack for a wider range of new applications on future hardware platforms.

1. INTRODUCTION

1.1 Data Analytics and Hardware Platforms

Data analytics

Data is growing exponentially. Every day, about 2.5 quintillion bytes of date is continuously being produced by data centers, humans, and Internet of Things (IoT) devices. For instance, we post half a million Tweets and generate four petabytes of data on Facebook per day. The trend of data growing will continue, reaching 463 exabytes of data per day in 2025 [1]. In order to extract useful insights from the large volume of data, data analytics has become one of the most important workloads.

Stream processing is one of the central paradigms of modern data analytics. Stream data consists of unbounded numbers of records. Each record includes one or more data values and a time stamp which indicates when the record was generated. The use cases of stream processing span many domains, such as tweet sentiment analysis, fraud detection in businesses, and log monitoring in data centers. Stream processing has an insatiable demand for high throughput and low latency.

Machine learning (ML) inference is another central paradigm of modern data analytics. It is the process of using a trained machine learning algorithm to make a prediction. The more data, the better the accuracy of its predictions. For example, object detection, traffic monitoring, and self driving all benefit from more data.

Hardware platforms

To meet the performance demands of data analytics, a large body of new hardware platforms are being shipped by hardware vendors. These new hardware platforms are very diverse and they target different performance goals, such as computation speed, memory bandwidth, and energy efficiency. Based on deployment scenarios, they can be classified into two categories.

Cloud hardware: Cloud is one of the most important platforms for data analytics. Due to the large volume of data, new hardware in cloud focus on improving computation speed

and memory speed. For instance, many-core CPUs have tens to hundreds of cores, which can speed up data analytics with high parallelism. 3D-stacked memory has 5 times higher bandwidth than general DRAM, so they can speed up data analytics by moving data faster.

Edge hardware: To avoid the costs of sending data to Cloud, there is a trend to push data analytics tasks to Edge IoT devices, driven by the new applications, such as smart homes, smart cities, and autonomous driving. On-device data analytics at the edge is attractive because it saves network bandwidth and preserves data privacy. Due to the constraints of cost and energy, IoT devices are resource-constraint and they focus on efficiency. For example, microcontrollers-based cameras and sensors have very tiny memory and battery.

1.2 Challenges to System Software (OS and Runtime)

The fast evolution in varying performance demands of data analytics applications and in varying new features of emerging hardware platforms brings significant challenges to existing system software, including operating systems and data analytics runtimes.

General operating systems are not aware of the unique characters and demands of data analytics applications. General OSes are designed for all kinds of applications and aim to achieve reasonable performance for all of them. They cannot adapt to the unique demands of one specific application. Therefore, the optimization space for specific data analytics in OSes is limited.

Existing data analytics systems are not aware of the unique features of underlying hardware. They are well optimized for fault tolerance, scalability, and consistency, like Spark Streaming and Apache Beam. They always try to add more machines to deal with more data, but neglect the optimization for single machine with new hardware features, which wastes resources.

1.3 Thesis Overview

The theme of this thesis is to provide systems support for data analytics by exploiting modern hardware. The goal of the thesis is to achieve high performance for data analytics on

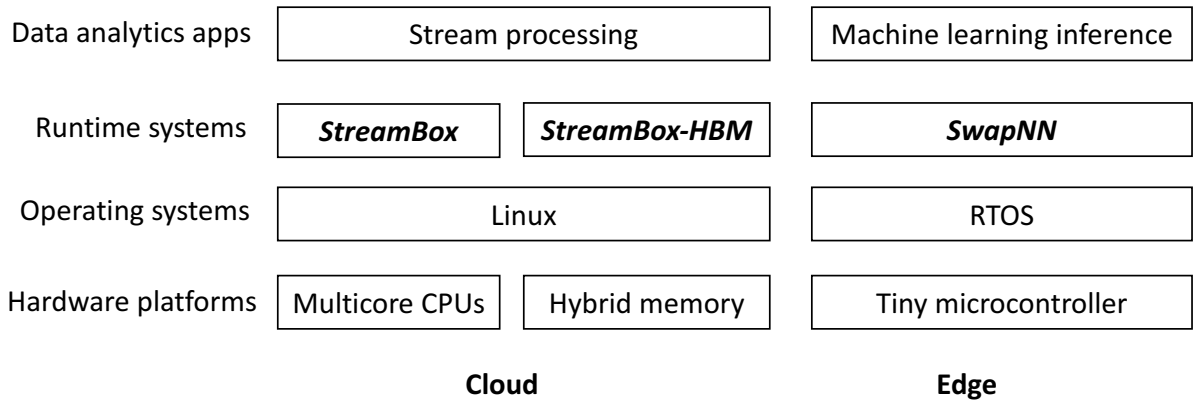


Figure1.1. An overview of the thesis work

advanced hardware platforms in the Cloud and enable on-device data analytics on resource-constraint hardware platforms at the Edge.

An overview of the thesis work is shown in Figure 1.1. We study two types of data analytics workloads, including stream processing and machine learning inference, on three types of hardware platforms across Cloud and Edge, including multicore CPUs, hybrid memory, and tiny microcontrollers.

Chapter 2 presents ***StreamBox*** [2], a stream analytics engine that exploits hardware parallelism of modern multicore machine, achieving scalable and highly efficient performance. We introduce a novel data structure called cascading containers to track dependences between epochs while at the same time processing any available records in any epoch. The key contribution of this work is a generalization of out-of-order record processing to out-of-order epoch processing that maximizes parallelism while minimizing synchronization overheads. Experimental results show our system scales to a large number of cores and out performs other state of the art engines. On the 56-core system, StreamBox reduces latency by a factor of 20 over Spark Streaming [3] and matches the throughput of results of Spark and Apache Beam [4] on medium-size clusters of 100 to 200 CPU cores for *grep* and *wordcount* queries.

Chapter 3 presents ***StreamBox-HBM*** [5], the first stream analytics engine that optimizes performance for hybrid memory combining high bandwidth memory (HBM) and DRAM, achieving memory scalability and highly efficient performance.. Our design addresses the limited capacity of HBM and HBM's need for sequential-access and high parallelism. We in-

troduce a novel dynamic key/record pointer extraction called KPA that minimizes the use of precious HBM capacity. We use sequential grouping algorithms on KPAs to balance limited capacity while exploiting high bandwidth. We design a runtime that manages parallelism and KPA placement in hybrid memories. Our system outperforms engines without KPA and with sequential-access algorithms by 7x and engines with random-access algorithms by an order of magnitude.

Chapter 4 presents ***SwapNN*** [6], the first system that enables high-accuracy neural network (NN) inference on extremely resource-constraint microcontrollers *without* losing accuracy, enabling new use cases and significantly expanding the scope of NN inference on tiny MCUs. Running neural networks (NNs) on microcontroller units (MCUs) is becoming increasingly important (e.g., tolerating poor networks and preserving data privacy), but is very difficult due to the tiny SRAM size of MCU. Prior work proposes many algorithm-level techniques to reduce NN memory footprints, but all at the cost of sacrificing accuracy and generality, which disqualifies MCUs for many important use cases. We investigate 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. Out-of-core NNs on MCUs raise multiple concerns: execution slowdown, storage wear out, energy consumption, and data security. We present a study showing that none is a showstopper; the key benefit – MCUs being able to run large NNs with full accuracy and generality – triumphs the overheads. Our findings suggest that MCUs can play a much greater role in edge intelligence.

Chapter 5 first summarizes the thesis work. It then presents the lessens we learned through building above systems and presents our general guidances on designing runtime systems across hardware/software stack for a wider range of new applications on future hardware platforms: (1) in application level, algorithms should adapt to hardware changes; (2) in runtime level, it's better managing resources than general hardware and OS, because runtime can leverage both applications' unique demand and hardware's unique features. (3) in OS level, kernel parameters should be configured accordingly to reduce OS overhead, e.g., enabling huge page and RDMA. (4) in hardware level, choosing hardware should be based on applications' demand.

2. STREAMBOX: STREAM ANALYTICS ON A MULTICORE MACHINE

2.1 Introduction

Stream processing is a central paradigm of modern data analytics. Stream engines process unbounded numbers of records by pushing them through a pipeline of *transforms*, a continuous computation on records [7]. Records have *event* timestamps, but they may arrive out-of-order, because records may travel over diverse network paths and computations on records may execute at different rates. To communicate stream progression, transforms emit timestamps called *watermarks*. Upon receiving a watermark w_{ts}, a transform is guaranteed to have observed all prior records with *event* time $\leq ts$.

Most stream processing engines are distributed because they assume processing requirements outstrip the capabilities of a single machine [3], [8], [9]. However, modern hardware advances make a single multicore machine an attractive streaming platform. These advances include (i) high throughput I/O that significantly improves ingress rate, e.g., Remote Direct Memory Access (RDMA) and 10Gb Ethernet; (ii) terabyte DRAMs that hold massive in-memory stream processing state; and (iii) a large number of cores. This work seeks to maximize streaming throughput and minimize latency on modern multicore hardware, thus reducing the number of required machines to process streaming workloads.

Stream processing on a multicore machine raises three major challenges. First, the streaming engine must extract parallelism aggressively. Given a set of transforms $\{d_1, d_2, \cdots, d_n\}$ in a pipeline, the streaming engine should exploit (i) pipeline parallelism by simultaneously processing all the transforms on different records in the data stream and (ii) data parallelism on all the available records in a transform. Second, the engine must minimize thread synchronization while respecting dependences. Third, the engine should exploit the memory hierarchy by creating sequential layout and minimizing data copying as records flow through various transforms in the pipeline.

Table2.1. Terminology

Term	Definition
Stream	An unbounded sequence of records
Transform	A computation that consumes and produces streams
Pipeline	A dataflow graph of transforms
Watermark	A special event timestamp for marking stream progression
Epoch	A set of records arriving between two watermarks
Bundle	A set of records in an epoch (processing unit of work)
Evaluator	A worker thread that processes bundles and watermarks
Container	Data structure that tracks watermarks, epochs, and bundles
Window	A temporal processing scope of records

2.2 Stream Model and Background

This section describes our out-of-order stream processing model and terminology, summarized in Table 2.1.

Streaming pipelines A stream processing engine receives one or more streams of *records* and performs a sequence of transforms $\mathcal{D} = \{d_1, d_2, \cdots, d_n\}$ on the records $\mathcal{R}$. Each record $r_{ts} \in \mathcal{R}$ has a timestamp ts for temporal processing. A record has an *event* timestamp defined by its occurrence (e.g., when a sensor samples a geolocation). Ingress of a record to the stream engine determines its *arrival* timestamp.

Out-of-order streaming Because data sources are diverse, records travel different paths, and transforms operate at different rates, records may arrive *out-of-order* at the stream processing engine or to individual transforms. To achieve low latency, the stream engine must *continuously* process records and thus cannot stall waiting for event and arrival time to align. We adopt the out-of-order processing (OOP) [10] paradigm based on *windows* to address this challenge.

Watermarks and stream epochs Ingress and transforms emit strictly monotonic event timestamps called *watermarks* w_{ts}, as exemplified in Figure 2.1(a). A watermark guarantees no subsequent records will have an event time earlier than ts. At ingress, watermarks delimit

ordered consecutive *epochs* of records. An epoch may have records with event timestamps greater than the epoch's end watermark due to out-of-order arrival. The stream processing engine may process records one at a time or in *bundles*.

We rely on stream sources and transforms to create watermarks based on their knowledge of the stream data [7], [11]. We do not inject watermarks (as does prior work [12]) to force output and manage buffering.

Pipeline egress Transforms define event-time *windows* that dictate the granularity at which to output results. Because we rely on watermarks to define streaming progression, the *rate* of egress is bounded by the rate of watermarks, since a transform can only close a window after it receives a watermark. We define the *output delay* in a pipeline from the time it first receives the watermark w_{ts} that signals the completion of the current window to the moment when it delivers the window results to the user. This critical path is implicit in the watermark timestamps. It includes processing any remaining records in epochs that precede w_{ts} and processing w_{ts} itself.

Programming model We use the popular model from timely dataflow [13], Google dataflow [7], and others. To compose a pipeline, developers declare transforms and define dataflows among transforms. This is exemplified by the following code that defines a pipeline for Windowed Grep, one benchmark used in our evaluation (§3.7).

```
// 1. Declare transforms
Source<string> source(/*config info*/);
FixedWindowInto<string> fwi(seconds(1));
WindowedGrep<string>wingrep(/*regexp*/);
Sink<string> sink();

// 2. Create a pipeline
Pipeline* p = Pipeline::create();
p->apply(source); //set source

// 3. Connect transforms together
connect_transform(source, fwi);
connect_transform(fwi, wingrep);
connect_transform(wingrep, sink);

// 4. Evaluate the pipeline
```

```
Evaluator eval(/*config info*/);
eval.run(p);  // run the pipeline
```

To implement a transform, developers must define the following functions, as shown in Figure 2.1(b): (i) `ProcessRecord(r)` consumes a record `r` and may emit derived records. (ii) `ProcessWm(w)` consumes a watermark `w`, flushes the transform's internal state, and may emit derived records and watermarks. `ProcessWm(w)` is always invoked only after `ProcessRecord(r)` consumes all records in the current epoch.

2.3 Design Goals and Criteria

We seek to exploit the potential of modern multicore hardware with its abundant hardware parallelism, memory capacity, and I/O bandwidth for high throughput and low latency. A key contribution of this work is exploiting *epoch parallelism* by concurrently processing all available epochs in every transform, in addition to pipeline parallelism. Epoch parallelism generalizes the idea of processing the records in each epoch out-of-order by processing epochs out-of-order. The following two invariants ensure correctness:

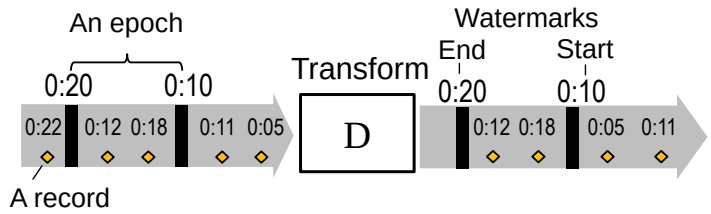
(1) *Records respect epoch boundaries* Each epoch is defined by a start watermark w_{start} and an end watermark w_{end} that arrive at ingress at time $start$ and end, and consists only of records r_{at} that arrive at ingress at time at, with $start < at < end$. Once an ingress record r_{at} is assigned an epoch, records never changed epochs, since this change might violate the watermark guarantee.

(2) *Watermark ordering* A transform D may only consume w_{end} after it consumes all the records r in the epoch. This invariant transitively ensures that watermarks and epochs are processed in order, and is critical to pipeline correctness, as it enforces the progression contract on ingress and between transforms.

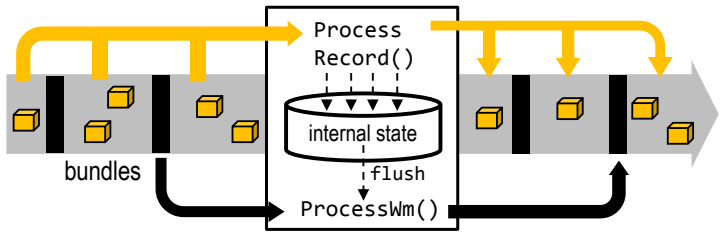
Our primary design goal is to minimize latency by exploiting epoch and pipeline parallelism with minimal synchronization while maintaining these invariants. In particular, our engine processes unconsumed records using all available hardware resources regardless of record ordering, delayed watermarks, or epoch ordering. We further minimize latency by exploiting the multicore memory hierarchy (i) by creating sequential memory layout and

minimizing data movement, and (ii) by mapping streaming data flows to the NUMA architecture.

2.4 System Overview



(a) A logic view of a transform, which consumes and produces out-of-order records as delimited by watermarks



(b) A transform in StreamBox. It is executed to consume and produce records belonging to multiple epochs in parallel.

Figure2.1. A transform in a StreamBox pipeline.

A StreamBox pipeline includes multiple transforms and each transform has multiple containers. Each container is linked to a container in a downstream transform or egress. Containers form a network pipeline organization, as depicted in Figure 2.2. Records, derived records, and watermarks flow through the network by following the links. A window consists of one or more epochs. The window size determines the output aggregation and memory layout, but otherwise does not influence how StreamBox manages epochs.

This dataflow pipeline network is necessary to exploit parallelism because parallelism emerges dynamically as a result of variation in record arrival times and the variation in processing times of individual records and watermarks for different transforms. For instance, records, based on their content, may require variable amounts of processing. Furthermore, it is typically faster to process a record than a watermark. However, exposing this abundant record processing parallelism and achieving low latency require prioritizing containers

on the critical path through the network. StreamBox prioritizes records in containers with timestamps preceding the pipeline's upcoming output watermark. Otherwise, the scheduler processes records from transforms with the most open containers. StreamBox thus dynamically adds parallelism to the bottleneck transforms of the network to optimize latency.

StreamBox implements three core components:

Elastic pipeline execution StreamBox dynamically allocates worker threads (*evaluators*) from a pool to transforms to maximize CPU utilization. StreamBox pins each evaluator to a CPU core to limit contention. During execution, StreamBox dispatches pending records and watermarks to evaluators. An evaluator executes transform code (i.e., `ProcessRecord()` or `ProcessWm()`) and produces new records and watermarks that further drive the execution of downstream transforms.

When dispatching records, StreamBox packs them into variable sized *bundles* for processing to amortize dispatch overhead and improve throughput. Bundles differ from batches in many other streaming engines [3], [9], [12]. First, bundle size is completely orthogonal to the transform logic and its windowing scheme. StreamBox is thus at liberty to vary bundle size dynamically per transform, trading dispatch latency for overhead. Second, dynamically packing records in bundles does not delay evaluators and imposes little buffering delay. StreamBox only produces sizable bundles when downstream transforms back up the pipeline.

Cascading containers Each container belongs to a transform and tracks one epoch, its state (*open, processing,* or *consumed*), the relationship between the epoch's records and its end watermark, and the output epoch(s) in the downstream consuming transform(s). Each transform owns a set of containers for its current input epochs. With this container state, executers may concurrently consume and produce records in *all* epochs without breaking or relaxing watermarks.

Pipeline state management StreamBox places records belonging to the same temporal windows (one or more adjacent epochs) in contiguous memory chunks. It adapts a bundle's internal organization of records, catering to data properties, e.g., the number of values per key. StreamBox steers bundles so that they flow mostly within their own NUMA nodes rather than across nodes. To manage transform internal state, StreamBox instantiates a contiguous

array of *slides* per transform, where each slide holds processing results for a given event-time range, e.g., a window. Evaluators operate on slide arrays based on window semantics, which are independent of the epoch tracking mechanism – cascading containers. The slide array realization incurs low synchronization costs under concurrent access.

2.5 Cascading Containers

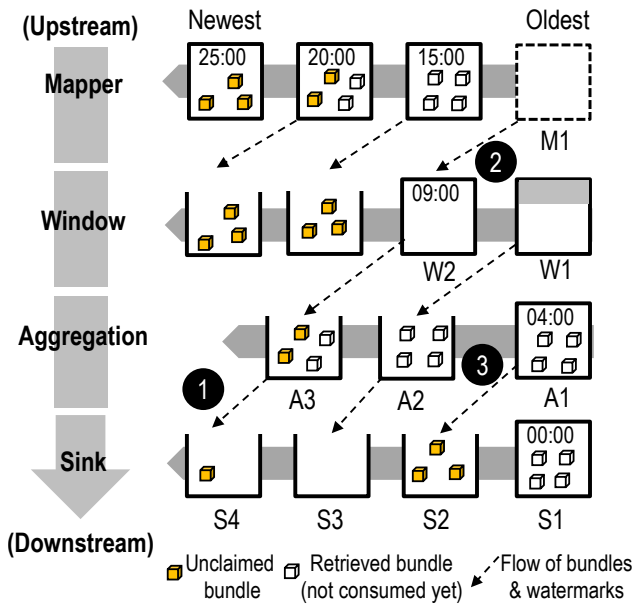


Figure2.2. An overview of cascading containers

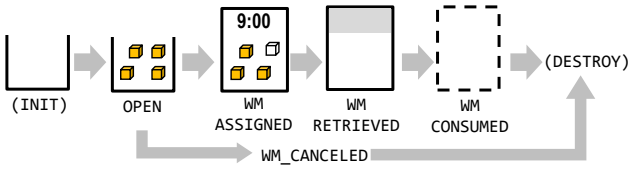


Figure2.3. The life cycle of a container

Cascading containers track epochs and orchestrate concurrent evaluators (i) to consume all of an epoch's records before processing its end watermark, (ii) to consume watermarks in stream order, and (iii) to emit records derived from an upstream epoch into the corresponding downstream epoch(s).

Figure 2.2 shows the cascading container design. Each transform owns a set of input stream containers, one for each potential epoch. When StreamBox creates a container *uc*, it

creates one downstream container dc (or more) for its output in the downstream transform(s) and links to it, causing a cascade of container creation. It puts all records and watermarks derived from the transform on uc into this corresponding downstream container dc. All these containers form a pipeline network. As stream processing progresses, the network topology evolves. Evaluators create new containers, establish links between containers, and destroy consumed containers.

2.5.1 Container Implementation

StreamBox initializes a container D_{own} when the transform receives the first input record or bundle of an epoch. Each container includes any unclaimed bundles of the epoch. An *unconsumed* counter tracks the number of bundles that ever entered the container but are not fully consumed. After processing a bundle, D_{own} deposits derived output bundles in the downstream container and then updates the unconsumed counter.

Container state StreamBox uses a container to track an epoch's life cycle as follows and shown in Figure 2.3.

open Containers are initially empty. An open container receives bundles from the immediate upstream D_{up}. The owner D_{own} processes the bundles simultaneously.

wm_assigned When D_{up} emits an epoch watermark w, it deposits w in D_{own}'s dependent container. Eventually D_{own} consumes all bundles in the container and the *unconsumed* counter drops to zero, at which point D_{own} retrieves and processes the end watermark.

wm_retrieved A container enters this state when D_{own} starts processing the end watermark.

wm_consumed After D_{own} consumes the end watermark, it guarantees that it has flushed all derived state and the end watermark to the downstream container and D_{own} may be destroyed.

wm_cancelled D_{up} chooses not to emit the end watermark for the (potential) epoch. Section 2.5.2 describes how we support windowing transforms by cancelling watermarks and merging containers.

Lock-free container processing Containers are lock-free to minimize synchronization overhead. We instantiate the end watermark as an atomic variable that enforces acquire-release memory order. It ensures that D_{own} observes all D_{up} evaluators' writes to the container's unclaimed bundle set before observing D_{up}'s write of the end watermark. The unclaimed bundle set is a concurrent data structure that aggressively weakens the ordering semantics on bundles for scalability. Examples of other such data structures include non-linearizable lock-free queues [14] and relaxed priority queues [15]. We further exploit this flexibility to make the bundle set NUMA-aware, as discussed in Section 2.7.1.

2.5.2 Single-input Transforms

If a transform has only one input stream, all its input epochs – and therefore the containers – are ordered, even though records are not.

Creating containers The immediate upstream container D_{up} creates downstream containers on-demand and links to them. Figure 2.2 ❶ shows an example of container creation. When StreamBox processes the first bundle in A3, it creates S4 and any missing container that precedes it, in this case S3, and links A3 to S4 and A2 to S3. To make concurrent growth safe, StreamBox instantiates downstream links and containers using an atomic variable with strong consistency. Subsequent bundle processing uses the instantiated links and containers.

Processing To select a bundle to process, evaluators walk the container list for a transform, starting from the oldest container to the youngest, since the oldest container holds the most urgent work for state externalization. If an evaluator encounters containers in the `wm_consumed` state, it destroys the container. Otherwise,

1. it retrieves an unclaimed bundle. If none exists,
2. it retrieves the end watermark when (i) the watermark is valid (i.e., the container has `wm_assigned`), and (ii) all bundles are consumed ($unconsumed == 0$), and (iii) all watermarks in the preceding containers of D_{own} are consumed.
3. If the evaluator fails to retrieve a bundle or watermark from this container, it moves to the next younger container on D_{own}'s list.

Figure 2.2 shows an example. An evaluator starts from the oldest container W1 to find work ❷. Because W1 is in WM_RETRIEVED (all bundles are consumed and the end

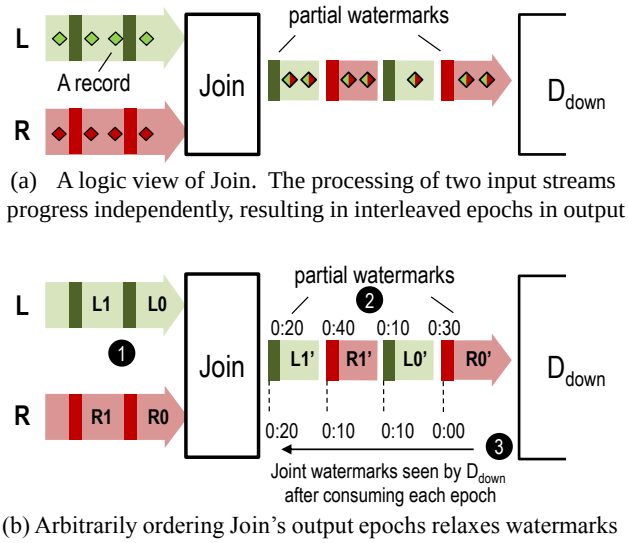


Figure2.4. A logic diagram of OOP temporal join

watermark is being processed), the worker moves on to W2. Because all bundles in W2 are consumed but the end watermark is available, it retrieves the watermark (09:00) for processing. Section 2.6 describes how we prioritize transforms in the container network.

Merging containers for windowing For each input container, we create a *potential* downstream container, expecting each input epoch will correspond to an output epoch. However, when a transform D performs windowing operations, it often must wait for multiple watermarks to correctly aggregate records. In this case, we merge containers. Figure 2.2 ❸ shows an example of Aggregation on a 10-min window. After consuming container A1 with its 04:00 watermark, the Aggregation transform cannot yet emit a watermark and retire its current window (0:00-10:00). Our solution is to cancel watermarks and merge the downstream output containers until the windowing logic, which uses event time, is satisfied. This operation is cheap. StreamBox cancels watermarks by simply marking them w_{cancel}. As evaluators walk container lists and observe w_{cancel}, they logically treat adjacent containers as one, e.g., S2 and S3. When the transform receives a watermark $ts \geq 10$, it emits the watermark which will eventually close the container.

2.5.3 Multi-input Transforms

A multi-input transform, such as temporal Join and Union, takes multiple input streams and produces one stream. Figure 2.4 shows an example of out-of-order temporal join [10]. The left and right input streams progress independently (they share D_{join}'s internal state). The output stream consists of *interleaved* epochs resulting from processing either input stream. These epochs are delimited by *partial* watermarks (w_L or w_R), which are also solely derived from the input streams. The downstream D_{down} derives a *joint* watermark as $min(w'_L, w'_R)$, where w'_L and w'_R are the most recent left and right partial watermarks.

The case for unordered containers A multi-input transform, unlike single-input transforms, cannot always have its downstream containers arranged on an *ordered* list (§2.5.2) because an optimal ordering of output epochs depends on their respective end (partial) watermarks. On the other hand, arbitrarily ordering output epochs may unnecessarily relax watermarks and delay watermark processing .

Figure 2.4(b) shows an example of arbitrarily ordering output epochs. While processing *open* input epochs L0/L1 and R0/R1 ❶, StreamBox arbitrarily orders the corresponding output as L1'→R1'→L0'→R0' without knowing the end watermarks. Later, these output epochs eventually receive their *partial* end watermarks ❷. Upon consuming them, D_{down} derives joint watermarks based on its subsequent observations of partial watermarks ❸. Unfortunately, the joint watermark is more relaxed than the partial watermarks. For instance, the partial watermark 00:30 of R0' guarantees that all records in R0' are later than 00:30. However, from the derived joint watermark, D_{down} only knows that they are later than 00:00. Relaxed watermarks propagate to all downstream transforms. To tighten a joint watermark, StreamBox should have placed L0' and L1' (and perhaps more subsequent left epochs) before R0' and R1'. However, it cannot make that decision before observing all these partial watermarks.

In summary, StreamBox must achieve two objectives in tracking epochs for multi-input transforms. (1) It must track output epochs with corresponding containers for epoch parallelism. (2) It must defer ordering these containers until it determines their end watermarks.

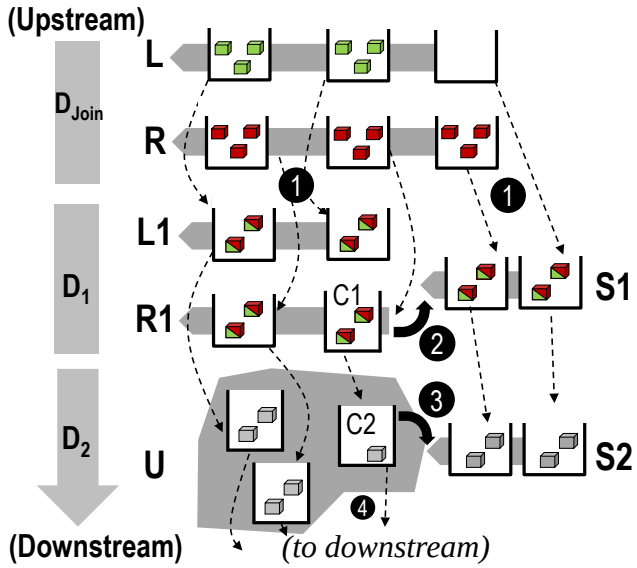


Figure2.5. Unordered containers for Join and its downstream. For brevity, container watermarks are not drawn

Solution StreamBox maintains *unordered* containers for a multi-input transform's output epochs and their downstream counterparts. Once StreamBox determines the ordering of one epoch, it appends the corresponding container to an ordered list and propagates this change downstream. Figure 2.5 shows an example.

- D_{join} owns two ordered container lists L and R.
- D_1, the immediate downstream transform of D_{join}, owns three ordered lists of containers. L1 and R1 are derived from D_{join}'s L and R, respectively. S1 holds merged containers from L1 and R1.
- With D_2 downstream of D_1, D_2 owns an unordered set U and an ordered list S2.

As D_{join} processes its input streams L and R, it deposits the derived bundles and watermarks to containers on L1, R1, and S1 ❶. D_1 selects the oldest container C1 on L1 and R1 to process and it appends C1 to S1 ❷. Processing C1, deposits records in container C2 (following the *down* link), which subsequently produces records in containers at S2 ❸ and beyond ❹.

2.5.4 Synchronized Access to Containers

In the cascading containers network, the concurrent evaluators dynamically modify the network topology by creating, linking, and destroying containers. Although the most frequent container operations, such as processing records, are lock-free as described in Section 2.5.1, modifying the container network must be synchronized. We carefully structure network modifications in reader and writer paths and synchronize them with one readers-writer lock for each container list. To retrieve work, an evaluator holds the container list's reader lock while walking the list. If the evaluator needs to modify the list (e.g., to destroy a container), it atomically upgrades the reader lock to a writer lock.

2.6 Pipeline Scheduling

A pipeline's latency depends on how fast the engine externalizes the state of the current window. To this end, StreamBox's scheduler prioritizes upcoming state externalization.

StreamBox maintains a global notion of the *next externalization moment (NEM)*. The upcoming windowed output requires processing of all bundles and watermarks with timestamps prior to NEM. After each state externalization, StreamBox increments the NEM monotonically based on a prediction. In the common case where externalization is driven by temporal windows, the engine can accurately predict NEM as the end of the current window. In case windowing information is unavailable, the engine may predict NEM based on historical externalization timing. Mispredicting NEM may increase the output delay but will not affect correctness.

NEM guides work prioritization in StreamBox. All evaluators independently retrieve work (i.e., bundles or watermarks) from cascading containers. By executing StreamBox's dispatch function, an evaluator looks for work by traversing container lists from the oldest to the youngest, starting from the top of the network. It prioritizes bundles in containers with timestamps that precede NEM.

Watermark processing is on the critical path of state externalization and often entails substantial amount of work, e.g., reduction of the window state. To accelerate watermark processing, StreamBox creates a special *watermark task queue*. Watermark tasks are defined

as lambda functions. StreamBox gives these tasks higher priority and executes them with the same set of evaluators – without oversubscribing the CPU cores. An evaluator first processes watermark tasks. After completing a task, evaluators return to the dispatcher immediately. Evaluators never wait on a synchronization barrier inside the watermark evaluator. This on-demand, extra parallelism accelerates watermark evaluation.

2.7 Pipeline State Management

The memory behavior of a stream pipeline is determined by the bundles of records flowing among transforms and the transforms' internal states. To manage this state, StreamBox targets locality, NUMA-awareness, and coarse-grained allocation/free. We decouple state management from other key aspects, including epoch tracking, worker scheduling, and transform logic.

2.7.1 Bundles

Adaptive internal structure StreamBox adaptively packs records into bundles for processing.

StreamBox seeks to (i) maximize sequential access, (ii) minimize data movement, and (iii) minimize the *per-record* overhead incurred by bundling.

A bundle stores a "flat" sequence of records sequentially in contiguous memory chunks. This logical record ordering supports grouping records temporally in epochs and windows, and by keys. It achieves both because temporal computation usually executes on all the keys of specific windows, rather than on specific keys of all windows. This choice contrasts to prior work that simply treats <window, key> as a new key.

To minimize data movement, StreamBox adapts bundle internals to the transform algorithm. For instance, given a Mapper that filters records, the bundles include both records and a bitmap, where each bit indicates the presence of a record, so that a record can be logically filtered by simply toggling a bit. Databases commonly use this optimization [12] as well.

StreamBox adapts bundle internals based on input data properties. The performance of keyed transforms, i.e., those consuming key-value pairs, is sensitive to the physical organization of these values. If each key has a large number of values, a bundle will hold a key's values using an array of pointers, each pointing to an array of values. This choice makes combining values produced by multiple workers as cheap as copying a few pointers. If each key only has a few values, StreamBox holds them in an array and copies them during combining. To learn about the input data, StreamBox samples a small fraction of it.

NUMA-aware bundle flows StreamBox explicitly steers bundles between transforms for NUMA locality by maximizing the chance that a bundle is both produced and consumed on the same NUMA node.

Each bundle resides in memory from one NUMA node and is labeled with that node. When an evaluator processes a container, it prefers unclaimed bundles labeled with its same NUMA node. It will process non-local bundles only when bundles from the local node are all consumed. To facilitate this process, an evaluator always allocates memory on its NUMA node, and later deposits the new bundle to the NUMA node of the downstream container. Notice that the NUMA-aware scheduling only affects the order among bundles within a container. It does not starve important work, e.g., containers to be dispatched by the next externalization moment.

2.7.2 Transform Internal State

StreamBox organizes a transform's internal state as an array of temporal *slides*, forming a slide. Each slide corresponds to a window (for fixed windows) or a window's offset (for sliding windows). Note that the size of a slide is independent of an epoch size.

To access a transform's state, an evaluator operates on a range of slides: updating slides in-place for accumulating processing results; fetching slides for closing a window; and retiring slides for state flushing. Since concurrent evaluators frequently access the slide arrays, we need to minimize locking and data movement. To achieve this goal, StreamBox grows the array on-demand and atomically. It only copies pointers when fetching slides. It decouples the logical retirement of slides from their actual, likely expensive destruction. To support

concurrent access to a single slide, the current StreamBox implementation employs off-the-shelf concurrent data structures, as discussed below.

2.8 Implementation

We implement StreamBox in 22K SLoC of C++11. The implementation extensively uses templates, static polymorphism, and C++ smart pointers. We implemented Windowing, GroupBy, Aggregation, Mapper, Reducer, and Temporal Join as our library transforms. Our scalable parallel runtime relies on the following scalable low-level building blocks.

C++ libraries We use boost [16] for timekeeping and locks, Intel TBB [17] for concurrent hash tables, and Facebook folly [18] for optimized vectors and strings. Folly improves the performance of some benchmarks by 20–30%. For scalable memory allocation, we use `jemalloc` [19], which scales much better than `std::alloc` and TBB [20] on our workloads.

Concurrent hash tables are hotspots in most statefull pipelines. We tested three open-source concurrent hash tables [17], [18], [21], but they either did not scale to a large core count or required pre-allocating a large amount of memory. Despite the extensive research on scalable hash tables [22], [23], we needed to implement an internally partitioned hash table. We wrapped TBB's concurrent hash map. This simple optimization improves our performance by 20–30%.

Bundle size is an empirical trade off between scheduling delay and overhead. StreamBox mainly varies bundle size at pipeline ingress. When the engine is fully busy, with all records in one ingress epoch, it produces as many bundles as evaluators, e.g., 56 bundles for 56 evaluators, to maximize the bundle size without starving any thread. The largest bundle size is around 80K records. When the ingress slows down, the system shrinks bundle sizes to reduce latency. We empirically determine that a 2× reduction in bundle size balances a 10% drop in ingress data rate. We set the minimal bundle size at 1K records to avoid excessive per-record overhead.

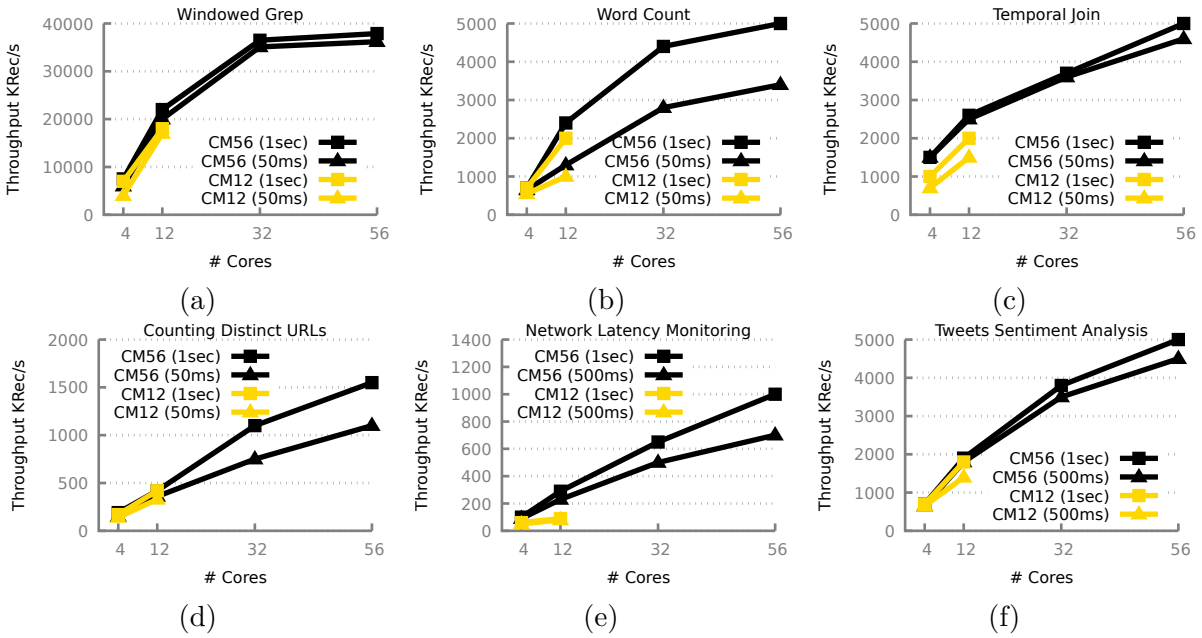


Figure2.6. Throughput of StreamBox as a function of hardware parallelism and latency. StreamBox scales well.

Table2.2. Test platforms used in experiments

56CM	*Dell PowerEdge R930* 4x14 Xeon E7-4850v4 "Broadwell", 256GB DRAM, Linux 4.4
12CM	*Dell PowerEdge R720* 2x6 Xeon E5-2630v2 "Ivy Bridge", 256GB DRAM, Linux 4.4

2.9 Evaluation

Methodology We evaluate StreamBox on the two multicore servers, summarized in Table 2.2. 56CM is a high-end server that excels at real-time analytics and 12CM is a mid-range server. Although 100 Gb/s Infiniband (RDMA) networks are available, our local network is only 10 Gb/s. However, 10 Gb/s is insufficient to test StreamBox and furthermore even if we used Infiniband, it will directly store stream input in memory. We therefore generate ingress streams from memory. We dedicate a small number of cores (1–3) to the pipeline source. We then replay these large memory buffers pre-populated with records and emit in-memory stream epochs continuously. We measure the maximum sustained throughput of up to 38 GB/s at the pipeline source when the pipeline delay meets a given target.

Benchmarks We use the following benchmarks and datasets. Unless stated otherwise, each input epoch contains 1 M records and spans 1 second of event time. (1) **Windowed Grep (grep)** searches the input text and outputs all occurrences of a specific string. We use Amazon Movie Reviews (8.7 GB in total) [24] as input, a sliding window of 30 seconds, and 1 second target latency. The input record size is 1 KB. (2) **Word Count (wordcount)** splits input texts into words and counts the occurrences of each word. We use 954 MB English books [25] as input, a sliding window of 30 seconds, and 1 second target latency. The input record size is 100 bytes. (3) **Temporal Join (join)** has two input streams, for which we randomly generate unique 64-bit integers as keys. The join window for each record is $\pm$ 0.5 seconds. (4) **Counting Distinct URLs (distinct)** [9] counts unique URL identifiers. We use the Yandex dataset [26] with 70 M unique URLs and a fixed window of 1 second. (5) **Network Latency Monitoring (netmon)** [9] groups network latency records by IP pairs and computes the average per group. We use the Pingmesh dataset [27] with 88 M records and a fixed window of 1 second. The source emits 500K records per epoch. (6) **Tweets Sentiment Analysis (tweets)** [9] correlates sentiment changes in a tweet stream to the most frequent words. It correlates results from two pipelines: one that selects windows with significant sentiment score changes, and the other that calculates the most frequent words for each window. We use a public dataset of 8 million English tweets [28] and a fixed window of 1 second. This benchmark is the most complex and uses 8 transforms.

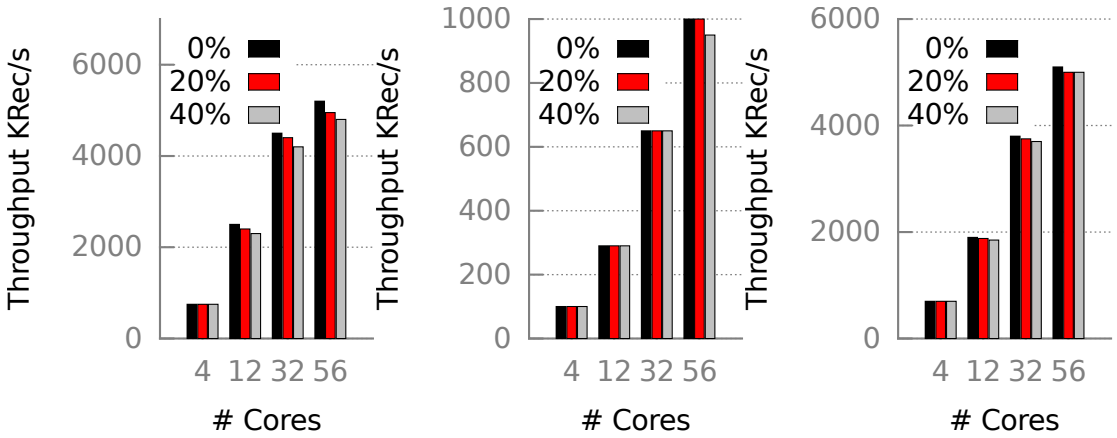
2.9.1 Throughput and Scalability

This section evaluates the throughput, scalability, and out-of-order handling of StreamBox, and compares with existing stream processing systems.

Throughput Figure 3.8 presents throughput on the y-axis for the six benchmarks as a function of hardware parallelism on the x-axis and latency as distinct lines. StreamBox has high throughput and typically processes millions of input records per second on a single machine, while delivering latencies as low as 50 ms. In particular, Grep achieves up to 38 M records per second, which translates to 38 GB per second. This outstanding performance is due to low overheads and high system utilization. Profiling shows that all CPU cores have

consistently high utilization (> 95%) and that most time is spent performing transform logic, e.g., processing stream data and manipulating hash tables.

Scalability Figure 3.8 shows that StreamBox scales well with core count for most benchmarks on both the 12-core and 56-core machines. When scalability diminishes in a few cases beyond 32 cores, as for Grep, it is a result of memory-bound computation saturating the machine.



(a) Wordcount (b) Netmon (c) Tweets

Figure2.7. StreamBox achieves high throughput even when a large fraction of records arrive out-of-order.

Out-of-order records By design, StreamBox efficiently computes on out-of-order records. To demonstrate this feature, we force a certain percent of records to arrive early in each epoch, i.e., the event time of these records is larger than the enclosing epoch's end watermark. Figure 2.7 shows the effect on throughput for 3 benchmarks. StreamBox achieves nearly the same throughput and latency as in in-order data processing. In particular, the throughput loss is as small as 7% even with 40% of records out-of-order. The minor degradation is due to early-arriving records that accumulate more windows in the pipeline. We attribute this consistent performance to (i) out-of-order epoch processing, since each transform continuously processes out-of-order records without delay, and (ii) prioritizing bundles and watermarks that decide the externalization latency of the current window in the scheduler.

Comparing to distributed stream engines We first compare StreamBox with published results of a few popular distributed stream processing systems and then evaluate two of

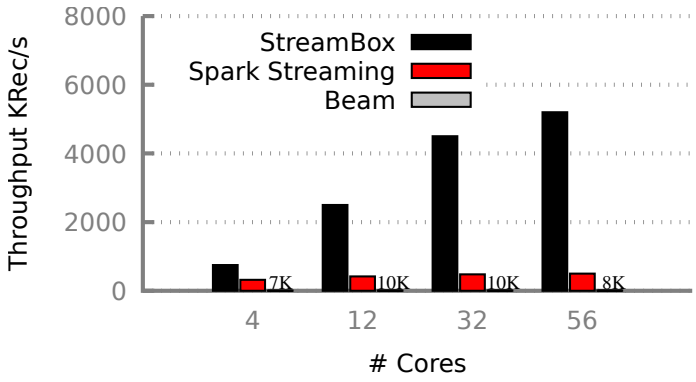


Figure2.8. StreamBox scales better than Spark and Beam with Wordcount on 56CM, with a 1-second target latency.

them on our 56-core machine. Most published results are based on processing of in-order stream data. For out-of-order data, they either lack support (e.g., no notion of watermarks) or expect transforms to "hold and sort", which significantly degrades latency [29], [30].

Compared to existing systems, StreamBox jointly achieves low millisecond latency and high throughput (tens of millions of records per second). Very few systems achieve both. To achieve similar throughput, prior work uses at least a medium-size cluster with a few hundred CPU cores [3], [8]. For instance, under the 50-ms target latency, StreamBox's throughput on 56CM is 40× greater than StreamScope [8] running on 100 cores. Moreover, even under a 1-second target latency, StreamBox achieves much higher throughput per core. StreamBox can process 700K records/sec for Grep and 90K records/sec for Wordcount per core, which are 4.7× and 1.5× faster than the per-core processing rate reported by Spark Streaming on a 100-node cluster with a total of 400 cores.

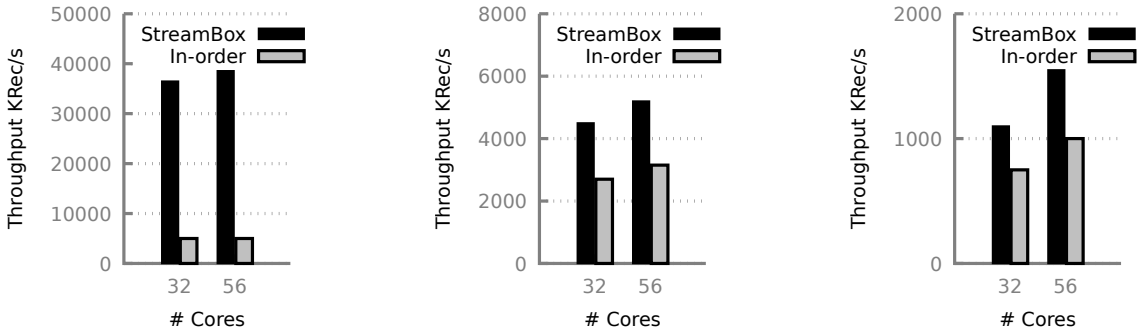
We further experimentally compare StreamBox with Spark (v2.1.0) [3] and Apache Beam (v0.5.0, executing its Direct Runner) [7], on the same machine (56CM). Note that Beam's Direct Runner is known to be unoptimized for a single machine. We verify that they both utilize all cores. We set the the target latency to 1 second since they cannot achieve 50 ms as StreamBox does. Figure 2.8 shows that StreamBox achieves significantly higher throughput (by more than one order of magnitude) and it scales much better with core count.

Comparing to single-machine streaming engines A few streaming engines are designed for a single machine: Oracle CEP [31], StreamBase [32], Esper [33], and SABER (for

CPU+GPU) [34]. With 4 to 16 CPU cores, they achieve throughput between thousands and a few million of records per second. None of them reports to scale beyond 32 CPU cores. In particular, we tested Esper [33] on 56CM with Wordcount. On four cores, Esper achieves around 900K records per second, which is similar to StreamBox with the same core count. However, we were unable to get Esper to scale even after applying recommended programming techniques, e.g., context partitioning [35]. As the core count increases, we observed the throughput drops.

In summary, StreamBox achieves better or similar per core performance than prior work. More importantly, StreamBox scales well to a large core count even with out-of-order record arrival.

2.9.2 Validation of Key Design Features

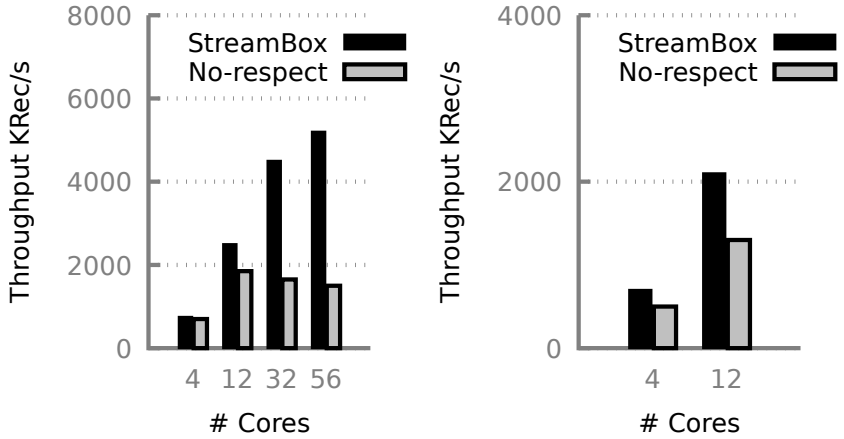


(a) Grep (b) Wordcount (c) Distinct

Figure2.9. In-order processing reduces parallelism, scalability, and throughput.

This section evaluates the performance and scaling contributions of our key design features.

Epoch parallelism for out-of-order processing Epoch parallelism is fundamental to producing abundant parallelism and exploiting out-of-order processing. We compare with in-order epoch processing by implementing "hold and sort," in which each transform waits to process an epoch until all its records arrive. Note that this in-order epoch processing leaves out the high cost of sorting records. It processes records within an epoch out-of-order.



(a) Wordcount on 56CM (b) Wordcount on 12CM

Figure2.10. When records do not respect epoch boundaries, it limits parallelism, scalability, and throughput.

Figure 2.9 shows that in-order epoch processing reduces throughput by 25% – 87%. Profiling reveals the reduced parallelism causes poor CPU utilization.

Records must respect epoch boundaries .StreamBox enforces the invariant that records respect epoch boundaries by mapping upstream containers to downstream containers (§2.5). We compare this to an alternative design where a transform's output records always flow into the *most recently* opened downstream container. Records then no longer respect epoch boundaries, since later records may enter earlier epochs. Violating the epoch invariant leads to huge latency fluctuations in watermark externalization, degrading performance. Figure 2.10 shows that not respecting epoch boundaries reduces throughput by up to 71%.

Prioritized scheduling (§2.6) Prioritizing containers on the critical path is crucial to latency and throughput. To explore its effect, we disable prioritized scheduling such that evaluators freely retrieve available bundles anywhere in the pipeline starting from its curent source and sink container. In this configuration, evaluators tend to rush into one transform, drain bundles there, and then move to the next. We confirmed this behavior with profiling. Performance measurements show that the pipeline latency fluctuates greatly and sometimes overshoots the target latency by a factor of 10.

NUMA-awareness (§2.7) We find NUMA-awareness especially benefits memory-bound benchmarks. For example, grep without windowing achieves 54 GB/s on 56CM, which is 12.5% higher than a configuration with NUMA-unaware evaluators.

Watermark arrival rates. Frequent watermarks lead to shorter epochs and more containers, each with fewer records, thus increasing the maintenance cost of cascading containers. In general, as shown in Figure 2.11, containers are sufficiently lightweight so that frequent watermarks (e.g., 100× more watermarks in 10K records/epoch) result in only a minor performance loss (e.g., 20%). However, substantial performance degradation emerges for watermarks at the rate of 1 K records/epoch, because frequent container creation and destruction incur too much synchronization.

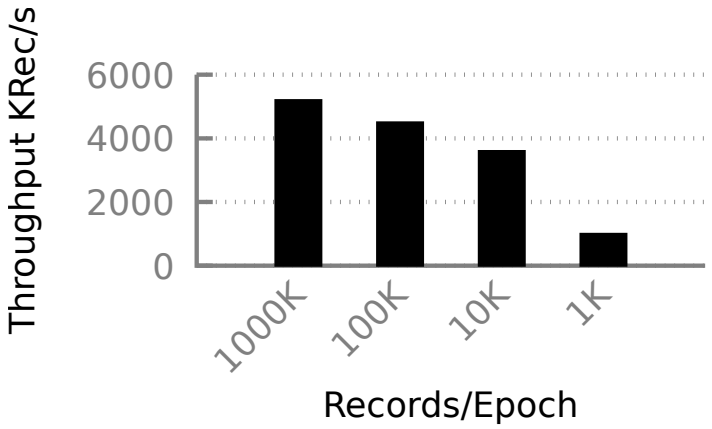


Figure2.11. Performance impact of watermark arrival rate for Wordcount on 56CM.

2.10 Related Work

This section compares StreamBox to prior work that uses the out-of-order processing (OOP) model, distributed and single server stream engines, and on exploiting shared memory for streaming.

OOP stream processing A variety of classic streaming engines focus on processing in-order records with a single core (e.g., StreamBase [32], Aurora [36], TelegraphCQ [37], Esper [33], Gigascope [29], and NiagaraST [38]). Li et al. [10] advocate OOP stream processing that relies on stream progression messages, e.g. punctuations, for better throughput and efficiency. The notion of punctuations is implemented in many modern streaming engines [7], [8], [12].

These systems do exploit pipeline and batch parallelism, but they do not exploit out-of-order processing of epochs to expose and deliver highly scalable data parallelism on a single server.

Single-machine streaming engines Trill [39] inspires StreamBox's state management with its columnar store and bit-vector design. However, Trill's punctuations are generated by the engine itself in order to flush its internal batches, which limits parallelism. Furthermore, Trill assumes ordered input records, which limits it applicability. StreamBox has neither of these limitations. SABER [34] is a hybrid streaming engine for CPUs and GPGPUs. Similar to StreamBox, it exploits data parallelism with multithreading. However, SABER does not support OOP. It must reorder execution results from concurrent workers, limiting its applicability and scalability. Oracle CEP [31] exploits record parallelism by relaxing record ordering. However, it lacks the notion of watermarks and does not implement statefull OOP pipelines.

Distributed streaming engines Several systems process large continuous streams using hundreds to thousands of machines. Their designs often focus on addressing the pressing concerns of a distributed environment, such as fault tolerance [3], [8], [9], programming models [7], [13], and API compatibility [40]. TimeStream [9] tracks data dependence between transform's input and output, but uses it for failure recovery. StreamBox also tracks fine-grained epoch dependences, but for minimizing externalization latency. StreamScope [8] handles OOP using watermark semantics, but it does not exploit OOP for performance as does StreamBox. It instead implements operator determinism based on holding and waiting for watermarks. StreamBox is partially inspired by Google's dataflow model [7] and is an implementation of its OOP programming model. However, to the best of our knowledge and based on our experiments, the Apache Beam [4] open-source implementation of Google dataflow does not exploit epoch parallelism on a multicore machine.

Data analytics on a shared memory machine Some data analytics engines propose to facilitate sequential memory access [41], [42] and one exploits NUMA **zhang2015numa**. StreamBox's bundles are similar to morsels in a relational query evaluator design [22], where evaluators process data fragments ("morsels") in batch and that are likely allocated on local NUMA nodes. StreamBox favors low scheduling delay for stream processing. Evaluators are

rescheduled after consuming each bundle, instead of executing the entire pipeline for that bundle.

2.11 Summary

StreamBox is an out-of-order stream processing engine for multicore machines. StreamBox organizes out-of-order records into *epochs* determined by *arrival time* at pipeline ingress and delimited by periodic *event time* watermarks. It manages all epochs with a novel parallel data structure called *cascading containers*. Each container manages an epoch, including its records and end watermark. StreamBox dynamically creates and manages multiple inflight containers for each transform. StreamBox links upstream containers to their downstream consuming containers. StreamBox provides three core mechanisms:

(1) StreamBox satisfies dependences and transform correctness by tracking producer/consumer epochs, records, and watermarks. It optimizes throughput and latency by creating abundant parallelism. It populates and processes multiple transforms and multiple in progress containers per transform. For instance, when watermark processing is a long latency event, StreamBox is not stalled, because as soon as any subsequent records arrive, it opens new containers and starts processing them.

(2) StreamBox elastically maps software parallelism to hardware. It binds a set of worker threads to cores. (i) Each thread independently retrieves a set of records (a *bundle*) from a container and performs the transform, producing new records that it deposits to a downstream container(s). (ii) To optimize latency, it prioritizes the processing of containers with timestamps required for the next stream output. As is standard in stream processing, outputs are scoped by *temporal windows* that are scoped by watermarks to one or more epochs.

(3) StreamBox judiciously places records in memory by mapping streaming access patterns to the memory architecture. To promote sequential memory access, it organizes pipeline state based on the output window size, placing records in the same windows contiguously. To maximize NUMA locality, it explicitly steers streams to flow within local NUMA nodes rather than across nodes.

We evaluate StreamBox on six benchmarks with a 12-core and a 56-core machines. StreamBox scales well up to 56 cores, and achieves high throughput (millions of records per second) and low latency (tens of milliseconds) on out-of-order records. On the 56-core system, StreamBox reduces latency by a factor of 20 over Spark Streaming [3] and matches the throughput of results of Spark and Apache Beam [7] on medium-size clusters of 100 to 200 CPU cores for *grep* and *wordcount*.

The full source code of StreamBox is available at http://xsel.rocks/p/streambox.

3. STREAMBOX-HBM: STREAM ANALYTICS ON HIGH BANDWIDTH HYBRID MEMORY

3.1 Introduction

Cloud analytics and the rise of the Internet of Things increasingly challenge stream analytics engines to achieve high throughput (tens of million records per second) and low output delay (sub-second) [2], [3], [12], [43]. Modern engines ingest unbounded numbers of time-stamped data records, continuously push them through a pipeline of operators, and produce a series of results over *temporal windows* of records. Many streaming pipelines group data in multiple rounds (e.g., based on record time and keys) and consume grouped data with a single-pass reduction (e.g., computing average values per key). For instance, data center analytics compute the distribution of machine utilization and network request arrival rate, and then join them by time. Data grouping often consumes a majority of the execution time and is crucial to low output delay in production systems such as Google Dataflow [44] and Microsoft Trill [12]. Grouping operations dominate queries in TPC-H (18 of 22) [45], BigDataBench (10 of 19) [46], AMPLab Big Data Benchmark (3 of 4) [47], and even Malware Detection [48]. These challenges require stream engines to carefully choose algorithms (e.g. Sort vs. Hash) and data structures for data grouping to harness the concurrency and memory systems of modern hardware.

Emerging 3D-stacked memories, such as high-bandwidth memory (HBM), offer opportunities and challenges for modern workloads and stream analytics. HBM delivers much higher bandwidth (several hundred GB/s) than DRAM, but at longer latencies and at reduced capacity (16 GB) versus hundreds of GBs of DRAM. Modern CPUs (KNL [49]), GPUs (NVIDIA Titan V [50]), FPGAs (Xilinx Virtex UltraScale+ [51]), and Cloud TPUs (v2 and v3 [52]) are using HBM/HBM2. Because of HBM capacity limitations, vendors couple HBM and standard DRAM in hybrid memories on platforms such as Intel Knights Landing [49]. Although researchers have achieved substantial improvements for high performance computing [53], [54] and machine learning [55] on hybrid HBM and DRAM systems, optimizing streaming for hybrid memories is more challenging. Streaming queries require high network bandwidth for ingress and high throughput for the whole pipeline. Streaming computations

are dominated by data grouping, which currently use hash-based data structures and random access algorithms. We demonstrate these challenges with measurements on Intel's Knights Landing architecture (§3.2). Delivering high throughput and low latency streaming on HBM requires high degrees of software and hardware parallelism and sequential accesses.

We present StreamBox-HBM, a stream analytics engine that transforms streaming data and computations to exploit hybrid HBM and DRAM memory systems. It performs sequential data grouping computations primarily in HBM. StreamBox-HBM dynamically extracts into HBM one set of keys at a time together with pointers to complete records in a data structure we call *Key Pointer Array* (KPA), minimizing the use of precious HBM memory capacity. To produce sequential accesses, we implement grouping computations as sequential-access parallel sort, merge, and join with wide vector instructions on KPAs in a streaming algorithm library. These algorithms are best for HBM and differ from hash-based grouping on DRAM in other engines [2], [3], [7], [56], [57].

StreamBox-HBM dynamically manages applications' streaming pipelines. At ingress, StreamBox-HBM allocates records in DRAM. For grouping computations for key k, it dynamically allocates extracted KPA records for k on HBM. For other streaming computations such as reduction, StreamBox-HBM allocates and operates on *bundles* of complete records stored in DRAM. Based on windows of records specified by the pipeline, the StreamBox-HBM runtime further divides each window into bundles to expose data parallelism in bottleneck stream operations. It uses bundles as the unit of computation, assigning records to bundles and threads to bundles or KPA. It detects bottlenecks and dynamically uses out-of-order data and pipeline parallelism to optimize throughput and latency by producing sufficient software parallelism to match hardware capabilities.

The StreamBox-HBM runtime monitors HBM capacity and DRAM bandwidth (the two major resource constraints of hybrid memory) and optimizes their use to improve performance. It prevents either resource from becoming a bottleneck with a single control knob: a decision on where to allocate new KPAs. By default StreamBox-HBM allocates KPAs on HBM. When the HBM capacity runs low, StreamBox-HBM gradually increases the fraction of new KPAs it allocates on DRAM, adding pressure to the DRAM bandwidth but without saturating it.

We evaluate StreamBox-HBM on a 64-core Intel Knights Landing with 3D-stacked HBM and DDR4 DRAM [58] and a 40 Gb/s Infiniband with RDMA for data ingress. On 10 benchmarks, StreamBox-HBM achieves throughput up to 110 M records/s (2.6 GB/s) with an output delay under 1 second. We compare StreamBox-HBM to Flink [57] on the popular YSB benchmark [59] where StreamBox-HBM achieves 18× higher throughput per core. Much prior work reports results without data ingress [2], [12]. As far as we know, StreamBox-HBM achieves the best reported records per second for streaming *with ingress* on a single machine.

The key contributions are as follows. (1) New empirical results find on real hardware that sequential sorting algorithms for grouping are best for HBM, in contrast to DRAM, where random hashing algorithms are best [60]–[62]. Based on this finding, we optimize grouping computations with sequential algorithms. (2) A dynamic optimization for limited HBM capacity that reduces records to keys and pointers residing in HBM. Although key/value separation is not new [12], [63]–[68], mostly it occurs statically ahead of time, instead of selectively and dynamically. (3) Our novel runtime manages parallelism and KPA placement based on both HBM's high bandwidth and limited capacity, and DRAM's high capacity and limited bandwidth. The resulting engine achieves high throughput, scalability, and bandwidth on hybrid memories. Beyond stream analytics, StreamBox-HBM's techniques should improve a range of data processing systems, e.g., batch analytics and key-value stores, on HBM and near-memory architectures [69]. To our knowledge, StreamBox-HBM is the first stream engine for hybrid memory systems. The full source code of StreamBox-HBM is available at http://xsel.rocks/p/streambox.

3.2 Background & Motivation

This section presents background on our stream analytics programming model, runtime, and High Bandwidth Memory (HBM). Motivating results explore `GroupBy` implementations with sorting and hashing on HBM. We find merge-sort exploits HBM's high memory bandwidth with sequential access patterns and high parallelism, achieving much higher throughput and scalability than hashing on HBM.

3.2.1 Modern Stream Analytics

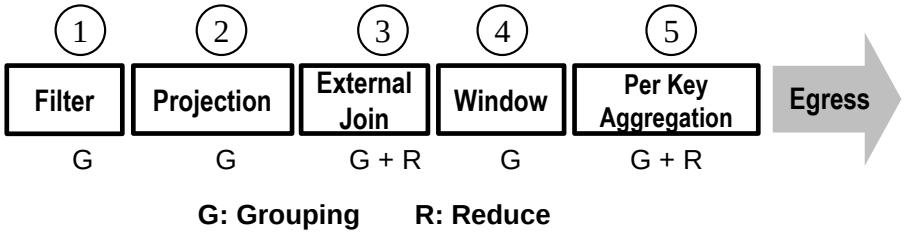
Programming model

We adopt the popular Apache Beam programming model [56] used by stream engines such as Flink [57], Spark Streaming [3], and Google Cloud Dataflow [7]. These engines all use declarative stream operators that group, reduce, or do both on stream data such as those in Table 3.3. To define a stream pipeline, programmers declaratively specify operators (computations) and a pipeline of how data flows between operators, as shown in the following pseudo code.

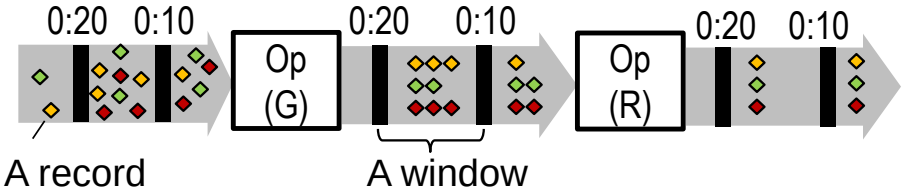
```
/* 1. Declare operators */
Source source(/* config info */);
WinGroupbyKey<key_pos> wingbk(1_SECOND);
SumPerKey<key_pos,v_pos> sum;
Sink sink;
/* 2. Create a pipeline */
Pipeline p; p.apply(source);
/* 3. Connect operators */
connect_ops(source, wingbk);
connect_ops(wingbk, sum);
connect_ops(sum, sink);
/* 4. Execute the pipeline */
Runner r( /* config info */ );
r.run(p);
```

Table3.1. Selected compound (declarative) operators in StreamBox-HBM and their constituent streaming primitives.

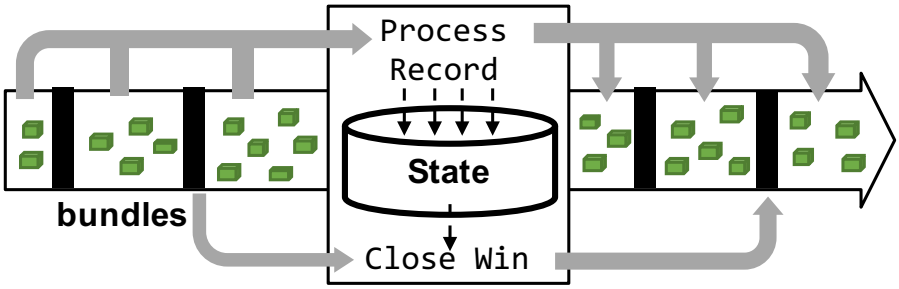
Compound Operators / Streaming Primitives on KPA	ParDo	AvgAll	Filter	Windowing	Union	CountByKey	TopKByKey	TempJoin	Cogroup
Grouping *Sort/Merge/Join…*			•	•	•	•	•	•	•
Reduction *Keyed/Unkeyed*	•	•				•	•	•	



(a) Pipeline of Yahoo streaming benchmark (YSB) which counts ad views. It filters records by ad_id 1, takes a projection on columns 2, joins by ad_id with associated campaign_id 3, then counts events per campaign per window 4 and 5. The pipeline will serve as our running example for design and evaluation



(b) A stream of records flowing through a grouping operator (G) and a reduction operator (R)



(c) Parallel operator execution. Engine batches records in bundles, consuming and producing bundles in multiple windows in parallel

Figure3.1. Example streaming data and computations

Streaming computations: grouping & reduction

The declarative operators in Table 3.3 serve two purposes. (1) *Grouping* computations organize records by keys and timestamps contained in sets of records. They sort, merge, or select a subset of records. Grouping may both move and compute on records, e.g., by comparing keys. (2) *Reduction* computations aggregate or summarize existing records and produce new ones, e.g., by averaging or computing distributions of values. Pipelines may interleave multiple instances of operations, as exemplified in Figure 3.1a. In most pipelines, grouping dominates total execution time.

Stream execution model

Figure 3.1b shows our execution model. Each stream is an unbounded sequence of records $\mathcal{R}$ produced by sources, such as sensors, machines, or humans. Each record consists of an event timestamp and an arbitrary number of attribute keys (*columns*). Data sources inject into record streams special *watermark* records that guarantee all subsequent record timestamps will be later than the watermark timestamp. However, records may arrive out-of-order [10]. A pipeline of stream operations consumes one or more data streams and generates output on temporal windows.

Stream analytics engine

Stream analytics engines are user-level runtimes that exploit parallelism. They exploit *pipeline parallelism* by executing multiple operators on distinct windows of records. We extend the StreamBox engine, which also exploits *data parallelism* by dividing windows into record bundles [2]. Figure 3.1c illustrates the execution of an operator. Multiple bundles in multiple windows are processed in parallel. After finishing processing one window, the runtime closes the window by combining results from the execution on each bundle in the window.

To process bundles, the runtime creates operator tasks, manages threads and data, and maps them to cores and memory resources. The runtime dynamically varies the parallelism

of individual operators depending on their workloads. At one given moment, distinct worker threads may execute different operators, or execute the same operator on different records.

3.2.2 Exploiting HBM

Modern HBM stacks up to 8 DRAM dies in special purpose silicon chips [70], [71]. Compared to normal DRAM, HBM offers (1) 5–10× higher bandwidth, (2) 5–10× smaller capacity due to cost and power [70]–[72], and (3) latencies typically ∼20% higher due to added stacking silicon layers.

Recent platforms couple HBM and DDR4-based DRAM as a hybrid memory system [49], [73], [74]. Hybrid memories with HBM and DRAM differ substantially from hybrid memories with SRAM and DRAM; or DRAM and NVM; or NUMA. In the latter systems, the faster tiers (e.g., on-chip cache or local NUMA memory) offer both higher bandwidth and *lower* latency. HBM lacks a latency benefit. We show next that for workloads to benefit from HBM, they must exhibit prodigious parallelism and sequential memory access *simultaneously*.

We measure two versions of GroupBy, a common stream operator on Intel's KNL with 96 GB of commodity DRAM and 16 GB of HBM (Table 3.4). (1) *Hash* partitions input ⟨key,value⟩ records and inserts them into an open-addressing, pre-allocated hash table. (2) *Sort* merge-sorts the input records by key (§ 3.4.2). We tune both implementations with hardware-specific optimizations and handwritten vector instructions. We derive our Hash from a state-of-the-art implementation hand-optimized for KNL [61], and implement Sort from a fast implementation [75] and hand-optimize it with AVX-512. Our *Hash* is 4× faster (not shown) than a popular, fast hash table *not* optimized for KNL [18]. Both implementations achieve state-of-the-art performance on KNL.

Figure 3.2 compares the throughput and bandwidth of *Sort* and *Hash* on HBM and DRAM. The x-axis shows the number of cores. We make the following observations. (1) *Sort* achieves the highest throughput and bandwidth when all cores participate. (2) When parallelism is low (fewer than 16 cores), the sequential accesses in *Sort* cannot generate enough memory traffic to fully exercise HBM high bandwidth, exhibiting throughput similar to *Sort* on DRAM. (3) HBM reverses the existing DRAM preference between *Sort* and

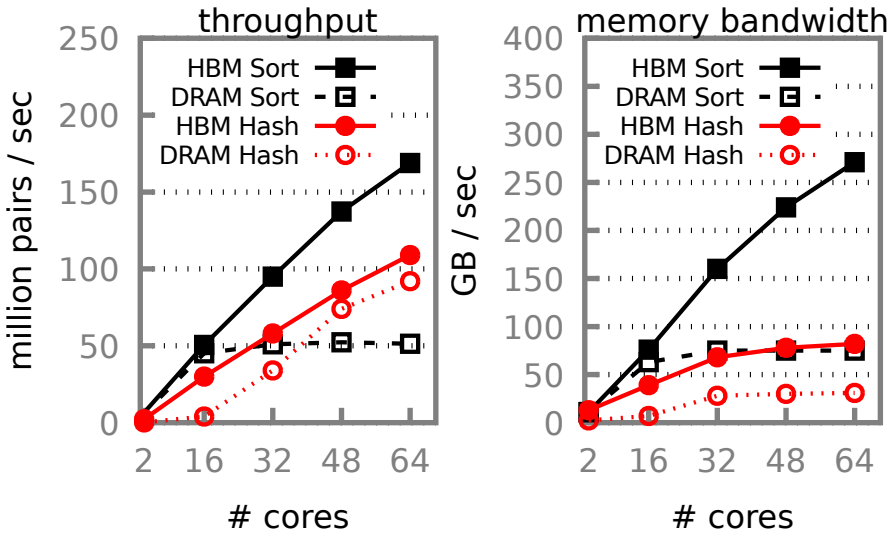


Figure3.2. GroupBy on HBM and DRAM operating on 100M key/value records with about 100 values per key. Keys and values are 64-bit random integers. Sort leverages HBM bandwidth with sequential access and outperforms Hash on HBM.

Hash. On DRAM, *Sort* is limited by memory bandwidth and underperforms *Hash* on more than 40 cores. On HBM, *Sort* outperforms *Hash* by over 50% for all core counts. *Hash* experiences limited throughput gains (10%) from HBM, mostly due to its sequential-access partitioning phase. *Sort*'s advantage over *Hash* is likely to grow as HBM's bandwidth continues to scale [72]. (4) HBM favors sequential-access algorithms even though they incur higher algorithmic complexity.

Prior work explored tradeoffs for *Sort* and *Hash* on DRAM [60]–[62], concluding *Hash* is best for DRAM. But our results draw a *different* conclusion for HBM – *Sort* is best for HBM. Because HBM employs a total wider bus (1024 bits vs. 384 bits for DRAM) with a wider SIMD vector (AVX-512 vs. standard AVX-256), it changes the tradeoff for software.

Why are existing engines inadequate?

Existing engines have shortcomings that limit their efficiency on hybrid memories. (1) Most engines use hash tables and trees, which poorly match HBM [2], [3], [12], [56], [57]. (2) They lack mechanisms for managing data and intermediate results between HBM and DRAM. Although the hardware or OS could manage data placement [76]–[78], their *reactive*

approaches use caches or pages, which are insufficient to manage the complexity of stream pipelines. (3) Stream workloads may vary over time due to periodic events, bursty events, and data resource availability. Existing engines lack mechanisms for controlling the resultant time-varying demands for hybrid memories. (4) With the exception of StreamBox [2], most engines generate pipeline parallelism, but do not generate sufficient total parallelism to saturate HBM bandwidth.

3.3 System Overview

We have three system design challenges: (1) creating sequential access in stream computations; (2) choosing which computations and data to map to HBM's limited capacity; and (3) trading off HBM bandwidth and limited capacity with DRAM capacity and limited bandwidth. To address them, we define a new smaller extracted data structure, new primitive operations, and a new runtime. This section overviews these components and subsequent sections describe them in detail.

Dynamic record extraction

StreamBox-HBM dynamically extracts needed keys and record pointers in a KPA data structure and operates on KPAs in HBM.

Sequential access streaming primitives

We implement data grouping primitives, which dominate stream analytics, with sequential-access parallel algorithms on numeric keys in KPAs. The reduce primitives dereference KPA pointers sequentially, randomly accessing records in DRAM, and operate on bundles of records in DRAM.

Plentiful parallelism

StreamBox-HBM creates computational tasks on KPA and bundles, producing sufficient pipeline and data parallelism to saturate the available cores.

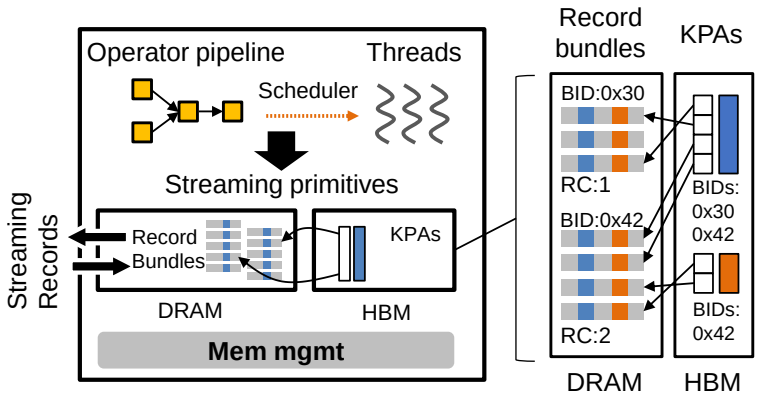


Figure3.3. An overview of StreamBox-HBM using record bundles and KPAs. RC: reference count; BID: bundle ID.

Dynamic mapping

When StreamBox-HBM creates a grouping task, it allocates or reuses a KPA in HBM. It monitors HBM capacity and DRAM bandwidth and dynamically balances their use by deciding where it allocates newly created KPAs. It never migrates existing data.

System architecture

StreamBox-HBM runs standalone on one machine or as multiple distributed instances on many machines. Since our contribution is the single-machine design, we focus the remaining discussion on one StreamBox-HBM instance. Figure 3.3 shows how StreamBox-HBM ingests streaming records through network sockets or RDMA and allocates them in DRAM – in arrival order and in row format. StreamBox-HBM dynamically manages pipeline parallelism similar to most stream engines [2], [3], [12], [57]. It further exploits data parallelism within windows with out-of-order bundle processing, as introduced by StreamBox [2].

3.4 KPA and Streaming Operations

This section first presents KPA data structures (§3.4.1) and primitives (§3.4.2). It then describes how KPAs and the primitives implement compound operators used by programmers

(§3.4.2), and how StreamBox-HBM executes an entire pipeline while operating on KPAs (§3.4.3).

3.4.1 KPA

To reduce capacity requirements and accelerate grouping, StreamBox-HBM extracts KPAs from DRAM and operates on them in HBM with specialized stream operators. Table 3.2 lists the operator API. KPAs are the *only* data structures that StreamBox-HBM places in HBM. A KPA contains a sequence of pairs of keys and pointers pointing to full records in DRAM, as illustrated in Figure 3.3. The keys replicate the record column required for performing the specified grouping operation without touching the full records. We refer to the keys in KPAs as *resident*. All other columns are *nonresident* keys.

One KPA represents intermediate grouping results. The first time StreamBox-HBM encounters a grouping operation on a key k, it creates a KPA by extracting the specified key for each record in one bundle and creating the pointer to the corresponding record. To execute a subsequent grouping computation on a new key q, StreamBox-HBM *swaps* the KPA's resident key with the new resident key q column for the corresponding record. After multiple rounds of grouping, one KPA may contain pointers in arbitrary order, pointing to records in arbitrary number of bundles, as illustrated in Figure 3.3. Each KPA maintains a list of bundles it points to, so that the KPA can efficiently update the bundles' reference counts. StreamBox-HBM reclaims record bundles after all the KPAs that point to them are destroyed.

Why one resident column?

We enclose *only one* resident column KPA because this choice greatly simplifies the implementation and reduces HBM memory consumption. We optimize grouping algorithms for a specific data type – key/pointer pairs, rather than for tuples with an arbitrary column count. Moving key/pointer pairs and swapping keys prior to each grouping operation is much cheaper than copying arbitrarily sized multi-column tuples.

Table3.2. KPA primitives. $\mathscr{R}$ denotes a record bundle. $KPA(c)$ denotes a KPA with resident keys from column c.

	Primitive	Access	Description
Maint	Extract $\mathscr{R} \rightarrow HBM(k)$	Sequential	Create a new KPA from a record bundle.
	Materialize $KPA(c) \rightarrow \mathscr{R}$	Random	Emit a bundle of full records according to KPA.
	KeySwap $KPA(c_1) \rightarrow KPA(c_2)$	Random	Replace a KPA's keys with a nonresident column.
Group	Sort $KPA(c) \rightarrow KPA(c)$	Sequential	Sort the KPA by resident keys
	Merge $KPA_1(c), KPA_2(c) \rightarrow KPA_3(c)$	Sequential	Merge two sorted KPAs by resident keys
	Join $KPA_1(c), KPA_2(c) \rightarrow \mathscr{R}$	Sequential	Join two sorted KPAs by resident keys. Emit new records.
	Select $\mathscr{R}$ or $KPA_1(c) \rightarrow KPA_2(c)$	Sequential	Subset a bundle as a KPA with surviving key/pointer pairs.
	Partition $KPA(c) \rightarrow \{KPA_{\mathrm{i}}(c)\}$	Sequential	Partition a KPA by ranges of resident keys.
Reduce	Keyed $KPA(c) \rightarrow \mathscr{R}$	Random	Do per-key reduction based on the resident keys.
	Unkeyed $\mathscr{R}_1$ or $KPA \rightarrow \mathscr{R}_2$	Random	Do reduction across all records.

3.4.2 Streaming Operations

Table3.3. Selected compound (declarative) operators in StreamBox-HBM and their constituent streaming primitives.

Streaming Primitives on KPA \ Compound Operators	ParDo	AvgAll	Filter	Windowing	Union	CountByKey	TopKByKey	TempJoin	Cogroup
Grouping *Sort/Merge/Join...*			•	•	•	•	•	•	•
Reduction *Keyed/Unkeyed*	•	•				•	•	•	

StreamBox-HBM implements the streaming primitives in Table 3.2, and the compound operators in Table 3.3. The primitives fall into the following categories.

- **Maintenance primitives** convert between KPAs and record bundles and swap resident keys. *Extract* initializes the resident column by copying the key value and initializing record pointers. *Materialize* and *KeySwap* scan a KPA and dereference the pointers. *Materialize* copies records to an output bundle in DRAM. *KeySwap* loads a nonresident column and overwrites its resident key.
- **Grouping primitives** *Sort* and *Merge* compare resident keys and rearrange key/pointer pairs within or across KPAs. Other primitives simply scan input KPAs and produce output in sequential order.
- **Reduction primitives** iterate through a bundle or KPA once and produce new records. They access nonresident columns with mostly random access. *Keyed* reduction scans a KPA, dereferences the KPA's pointers, locates full records, and consumes nonresident column(s). Per-key aggregation scans a sorted KPA and keeps track of contiguous key ranges. For each key range, it coalesces values from a nonresident column. *Unkeyed* reduction scans a record bundle, consumes nonresident column(s), and produces a new record bundle.

Primitive Implementation

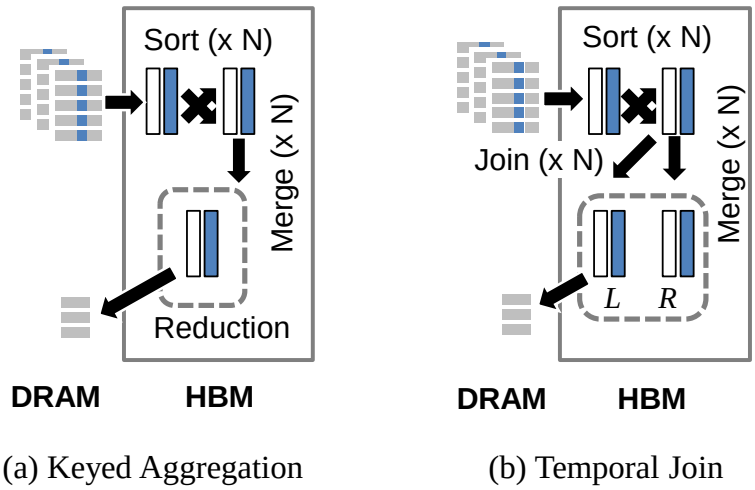
Our design goal for primitive operations is to ensure that they all have high parallelism and that grouping primitives produce sequential memory access. All primitives operate on 64-bit value key/pointer pairs. They compare keys and based on the comparison, move keys and the corresponding pointers.

Our *Sort* implementation is a multi-threaded merge-sort. It first splits the input KPA into N chunks, sorts each chunk with a separate thread, and then merges the N sorted chunks. A thread sorts its chunk by splitting the chunk into *blocks* of 64× 64-bit integers, invoking a bitonic sort on each block, and then performing a bitonic merge. We hand-tuned the bitonic sort and merge kernels with AVX-512 instructions for high data parallelism. After sorting chunks, all N threads participate in pairwise merge of these chunks iteratively. As the count of resultant chunks drops below N, the threads slice chunks at key boundaries to parallelize the task of merging fewer, but larger chunks among them. *Merge* reuses the parallel merge logic in *Sort*. *Join* first sorts the input KPAs by the *join* key. It then scans them in one pass – comparing keys and emitting records along the way.

Compound Operators

We implement four common families of compound operators with streaming primitives and KPAs.

- **ParDo** is a stateless operator that applies the same function to every record, e.g., filtering a specific column. StreamBox-HBM implements *ParDo* by scanning the input in sequential order. If the ParDo does not produce new records (e.g., *Filter* and *Sample*), StreamBox-HBM performs *Selection* over KPA. When they produce new records (e.g., *FlatMap*), StreamBox-HBM performs *Reduction* and emits new records to DRAM.
- **Windowing** operators group records into temporal windows using *Partition* on KPA. They treat the timestamp column as the partitioning key and window length (for fixed windows) or slide length (for sliding windows [79]) as the key range of each output partition.



(a) Keyed Aggregation (b) Temporal Join

Figure3.4. Declarative operators implemented atop KPAs

- **Keyed Aggregation** is a family of statefull operators that aggregate given column(s) of the records sharing a key (e.g., *AverageByKey* and *PercentileByKey*). StreamBox-HBM implements them using a combination of *Sort* and *Reduction* primitives, as illustrated in Figure 3.4a. As N bundles of records in the same window arrive, the operator extracts N corresponding KPAs, sorts the KPAs by key, and saves the sorted KPAs as internal state for the window (shown in the dashed-line box). When the operator observes the window's closure by receiving a watermark from upstream, it merges all the saved KPAs by key k. The result is a KPA(k) representing all records in the window sorted by k. The operator then executes per-key aggregation as out-of-KPA reduction as discussed earlier. The implementation performs each step in parallel with all available threads. As an optimization, the threads perform early aggregation on individual KPAs before the window closure.
- **Temporal Join** takes two record streams L and R. If two records, one in L and one in R in the same temporal window, share a key, it emits a new combined record. Figure 3.4b shows the implementation for R. For the N input bundles in R, StreamBox-HBM extracts their respective KPAs, sorts the KPAs, and performs two types of primitives in parallel: (1) *Merge*: the operator merges all the sorted KPAs by key. The resultant KPA is the window state for R, as shown inside the dashed line box of the figure. (2) *Join* with L: in parallel with *Merge*, the operator joins each of the aforementioned sorted KPA with the window state on

L shown in the dashed line box. StreamBox-HBM concurrently performs the same procedure on L. It uses primitive *Join* on two sorted KPA(k)s, which scans both in one pass. The operator emits to DRAM the resultant records, which carry the join keys and any additional columns.

3.4.3 Pipeline Execution Over KPAs

During pipeline execution, StreamBox-HBM creates and destroys KPA and swaps resident keys dynamically. It seeks to execute grouping operators on KPA and minimize the number of accesses to nonresident columns in DRAM. At pipeline ingress, StreamBox-HBM ingests full records into DRAM. Prior to executing any primitive, StreamBox-HBM examines it and transforms the input of grouping primitives as follows.

```
/* X: input (a KPA or a bundle) */
/* c: column containing grouping key */
X = IsKPA(X) ? X : Extract(X)
if ResidentColumn of X != c
  KeySwap(X, c)
Execute grouping on X
```

StreamBox-HBM applies a set of optimizations to further reduce the number of DRAM accesses. (1) It coalesces adjacent *Materialize* and *Extract* primitives to exploit data locality. As a primitive emits new records to DRAM, it simultaneously extracts the KPA records required by the next operator in the pipeline. (2) It updates KPA's resident keys in place, and writes back dirty keys to the corresponding nonresident column as needed for future *KeySwap* and *Materialize* operations. (3) It avoids extracting records that contain fewer than three columns, which are already compact.

Example

We use YSB [59] in Figure 3.1a to show pipeline execution. We omit *Projection*, since StreamBox-HBM stores results in DRAM. Figure 3.5 shows the engine ingesting record bundles to DRAM ①. Filter, the first operator, scans and selects records based on column *ad_type*, producing KPA(*ad_id*) ②. *External Join* (different from temporal join) scans the KPA and updates the resident keys *ad_id* in place with *camp_id* loaded from an external

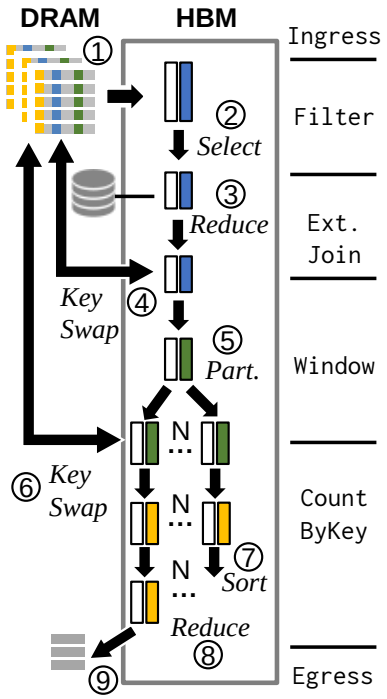


Figure3.5. Pipeline execution on KPAs for YSB [59]. Declarative operators shown on right.

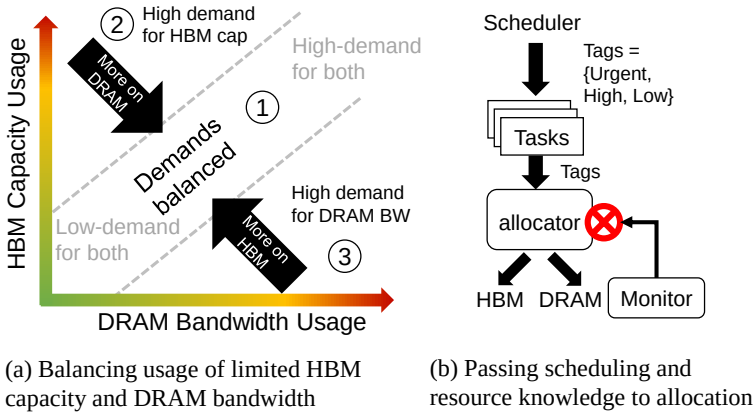
key-value store ③, which is a small table in HBM. The operator writes back *camp_id* to full records and swaps in timestamps t ④, resulting in KPA(t). Operator *Window* partitions the KPA by t ⑤. *Keyed Aggregation* swaps in the grouping key *camp_id* ⑥, sorts the resultant KPA(*camp_id*) ⑦, and runs reduction on KPA(*camp_id*) to count per-key records ⑧. It emits per-window, per-key record counts as new records to DRAM ⑨.

3.5 Dynamically Managing Hybrid Memory

In spite of the compactness of KPAs representation, HBM still cannot hold all the KPAs at once. StreamBox-HBM manages *which* new KPAs to place on *what* type of memory by addressing the following two concerns.

1. *Balancing demand.* StreamBox-HBM balances the aggregated demand for limited HBM capacity and DRAM bandwidth to prevent either from becoming a bottleneck.

2. *Managing performance.* As StreamBox-HBM dynamically schedules a computation, it optimizes for the access pattern, parallelism, and contribution to the critical path by where



(a) Balancing usage of limited HBM capacity and DRAM bandwidth

(b) Passing scheduling and resource knowledge to allocation

Figure3.6. StreamBox-HBM dynamically manages hybrid memory

it allocates the KPA for the computation. StreamBox-HBM prioritizes creating KPA in HBM for aggregation operations on the critical path to pipeline output. When work is on the critical path, it further prioritizes increasing parallelism and throughput for these operations versus KPA that are processing early arriving records. We mark bundles an *urgent* on the critical path with a *performance impact tag*, as described below.

StreamBox-HBM monitors HBM capacity and DRAM bandwidth and trades them off dynamically. For individual KPA allocations, StreamBox-HBM further considers the critical path. StreamBox-HBM does not migrate existing KPAs, which are ephemeral, unlike other software systems for hybrid memory [76]–[78].

Dynamically Balancing Memory Demand

Figure 3.6 plots StreamBox-HBM's state space. StreamBox-HBM strives to operate in the diagonal zone ①, where limiting capacity and bandwidth demands are balanced. If both capacity and bandwidth reach their limit, StreamBox-HBM operates in the top-right corner in zone ①, while throttling the number of concurrent threads working on DRAM to avoid over-subscribing bandwidth and wasting cores, and preventing back pressure on ingestion.

When the system becomes imbalanced, the state moves away from zone ① to ② or ③. Example causes include additional tasks spawned for DRAM bundles which stress DRAM bandwidth, and delayed watermarks that postpone window closure which stresses HBM

capacity. If left uncontrolled, such imbalance will lead to performance degradations. When HBM is full, all future KPAs regardless of their performance impact tag are forced to spill to DRAM. When DRAM bandwidth is fully saturated, additional parallelism on DRAM wastes cores.

At runtime, StreamBox-HBM balances resources by tuning a global *demand balance knob* as shown in Figure 3.6. StreamBox-HBM gradually changes the fraction of the *new* KPA allocations on HBM or DRAM, and pushes its state back to the diagonal zone. In rare cases, there is no more HBM capacity and no more DRAM bandwidth because the data ingestion rate is too high. To address this issue, StreamBox-HBM dynamically starts or stops pulling data from data source according to current resource utilization.

Performance impact tags

To identify the critical path, StreamBox-HBM maintains a global target watermark, which indicates the next window to close. StreamBox-HBM deems any records with timestamps earlier than the target watermark on the critical path. When creating a task, the StreamBox-HBM scheduler tags it with one of three coarse-grained *impact* tags based on when the window that contains the data for this task will be externalized. Windows are externalized based on their record-time order. (1) *Urgent* is for tasks on the critical path of pipeline output. Examples include the last task in a pipeline that aggregates the current window's internal state. (2) *High* is for tasks on younger windows (i.e., windows with earlier record time), for which results will be externalized in the *near* future, say one or two windows in the future. (3) *Low* is for tasks on even younger windows, for which results will be externalized in the *far* future.

Demand balance knob

We implement a demand balance knob as a global vector of two scalar values $\{k_{low}, k_{high}\}$, each in the range of $[0, 1]$. k_{low} and k_{high} define the probabilities for StreamBox-HBM to allocate KPAs on HBM for Low and High tasks correspondingly. *Urgent* tasks always allocate

KPAs from a small reserved pool of HBM. The knob in conjunction with each KPA allocation's performance impact tag determines the KPA placement as follows.

```
/* to choose memory type to be M */
switch (alloc_perf_tag)
case Urgent:
  M = HBM
case High:
  M = random(0,1) < k_high ? HBM : DRAM
case Low:
  M = random(0,1) < k_low ? HBM : DRAM
allocate on M
```

StreamBox-HBM refreshes the knob values every time it samples the monitored resources. It changes the knob values in small increments Δ for controlling future HBM allocations. To balance memory demand it first considers changing k_{low}; if k_{low} already reaches an extreme (0 or 1), StreamBox-HBM considers changing k_{high} if the pipeline's current output delay still has enough headroom (10%) below the target delay. We set the initial values of k_{high} and k_{low} to 1, and set Δ to 0.05.

3.5.1 Memory Management and Resource Monitoring

StreamBox-HBM manages HBM memory with a custom slab allocator on top of a memory pool with different fixed-sized elements, tuned to typical KPA sizes, full record bundle sizes, and window sizes. The allocator tracks the amount of free memory. StreamBox-HBM measures DRAM bandwidth usage with Intel's processor counter monitor library [80]. StreamBox-HBM samples both metrics at 10 ms intervals, which are sufficient for our analytic pipelines that target sub-second output delays.

By design, StreamBox-HBM never modifies a bundle by adding, deleting, or reordering records. After multiple rounds of grouping, all records in a bundle may be dead (unreferenced) or alive but referenced by different KPAs. StreamBox-HBM reclaims a bundle when no KPA refers to any record in the bundle using reference counts (RC). On the KPA side, each KPA maintains one reference for each source bundle to which any record in the KPA points. On the bundle side, each bundle stores a reference count (RC) tracking how many KPAs link to it. When StreamBox-HBM extracts a new KPA ($\mathscr{R} \rightarrow KPA$), it adds a link

pointing to $\mathscr{R}$ if one does not exist and increments the reference count. When it destroys a KPA, it follows all the KPA's links to locate source bundles and decrements their reference counts. When merging or partitioning KPAs, the output KPA(s) inherits the input KPAs' links to source bundles, and increments reference counts at all source bundles. When the reference count of a record bundle drops to zero, StreamBox-HBM destroys the bundle.

3.6 Implementation and Methodology

We implement StreamBox-HBM in C++ atop StreamBox, an open-source research analytics engine [2], [81]. StreamBox-HBM has 61K lines of code, of which 38K lines are new for this work. StreamBox-HBM reuses StreamBox's work tracking and task scheduling, which generate task and pipeline parallelism. We introduce new operator implementations and novel management of hybrid memory, replacing all of the StreamBox operators and enhancing the runtime, as described in the previous sections. The current implementation supports numerical data, which is very common in data analytics [82].

Benchmarks

We use 10 benchmarks with a default window size of 10 M records that spans one second of event time. One is YSB, a widely used streaming benchmark [43], [83], [84]. YSB processes input records with seven columns, for which we use numerical values rather than JSON strings. Figure 3.1a shows its pipeline.

We also use nine benchmarks with a mixture of widely tested, simple pipelines (1–8) and one complex pipeline (9). All benchmarks process input records with three columns – keys, values, and timestamps, except that input records for benchmark 8 and 9 contain one extra column for secondary keys. (1) **TopK Per Key** groups records based on a key column and identifies the top K largest values for each key in each window. (2) **Windowed Sum Per Key** aggregates input values for every key per window. (3) **Windowed Median Per Key** calculates the median value for each key per window. (4) **Windowed Average Per Key** calculates the average of all values for each key per window. (5) **Windowed Average All** calculates the average of all values per window. (6) **Unique Count Per Key** counts unique

values for each key per window. (7) **Temporal Join** joins two input streams by keys per window. (8) **Windowed Filter** takes two input streams, calculates the value average on one stream per window, and uses the average to filter the key of the other stream. (9) **Power Grid** is derived from a public challenge [85]. It finds houses with the most high-power plugs. Ingesting a stream of per-plug power samples, it calculates the average power of each plug in a window and the average power over all plugs in all houses in the window. Then, for each house, it counts the number of plugs that have higher load than average. Finally, it emits the houses that have most high-power plugs in the window.

For YSB, we generate random input following the benchmark directions [59]. For Power Grid, we replay the input data from the benchmark [85]. For other benchmarks, we generate input records with columns as 64-bit random integers. Note that our grouping primitives, e.g. sort and merge, are insensitive to key skewness [86].

Hardware platform

We implement StreamBox-HBM on KNL [58], a manycore machine with hybrid HBM/-DRAM memory. Compared to the standard DDR4 DRAM on the machine, the 3D-stacked HBM DRAM offers 5× higher bandwidth with 20% longer latency. The machine has 64 cores with 4-way simultaneous multithreading for a total of 256 hyper-threads. We launch one thread per core as we find out this configuration outperforms two or four hyper-threads per core due to the number of outstanding memory requests supported by each core. The ISA includes AVX-512, Intel's wide vector instructions. We set BIOS to configure HBM and DRAM in *flat* mode, where both memories appear fully addressable to StreamBox-HBM. We also compare to *cache* mode, where HBM is a hardware-managed last-level cache in front of the DDR4 DRAM. Table 3.4 summarizes the KNL hardware and a 56-core Intel Xeon server (X56) used in evaluation for comparisons.

Data ingress

We use a separate machine (an i7-4790 with 16 GB DDR4 DRAM) called Sender to generate input streams. To create sufficient ingestion bandwidth, we connect Sender to KNL

Table3.4. KNL and Xeon Hardware used in evaluation

KNL	Xeon Phi 7210	**$5,000**
CPU:	64 Cores @ 1.3 GHz	
HBM:	16 GB BW: 375 GB/s Latency: 172 ns	
DRAM:	DDR4 96 GB BW: 80 GB/s Latency: 143 ns	
NIC1:	40Gb/s Infiniband Mellanox ConnectX-2	
NIC2:	10GbE Mellanox ConnectX-2	
X56	Xeon E7-4830v4 "Broadwell"	**$23,000**
CPU:	4x14 cores @ 2.0 GHz	
DRAM:	DDR4 256 GB BW: 87 GB/s Latency: 131 ns	
NIC:	10GbE Intel X540 DP	

using RDMA over 40 Gb/s Infiniband. With RDMA ingestion, StreamBox-HBM on KNL pre-allocates a pool of input record bundles. To ingest bundles, StreamBox-HBM informs Sender of the bundle addresses and then polls for a notification which signals bundle delivery from Sender. To compare StreamBox-HBM with commodity engines that do not support RDMA ingestion, we also deliver input over our available 10 Gb/s Ethernet using the popular, fast ZeroMQ transport [87]. With ZeroMQ ingestion, the engine copies incoming records from network messages and creates record bundles in DRAM.

3.7 Evaluation

We first show StreamBox-HBM outperforms Apache Flink [57] on YSB. We then evaluate StreamBox-HBM on the other benchmarks, where it achieves high throughput by exploiting high memory bandwidth. We demonstrate that the key design features, KPA and dynamically balancing memory and performance demands, are essentially to achieving high throughput.

3.7.1 Comparing to Existing Engines

Comparing to Flink on YSB

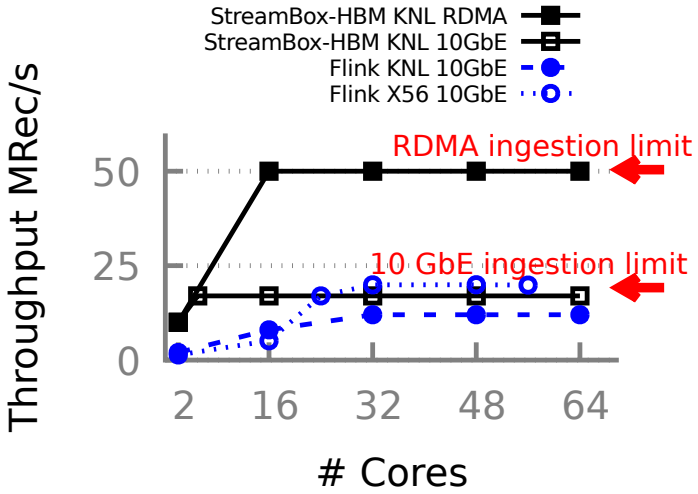
We compare to Apache Flink (1.4.0) [57], a popular stream analytics engine known for its good single-node performance on the YSB benchmark described in Section 3.6. To compare fairly, we configure the systems as follows. (1) Both StreamBox-HBM and Flink ingest data using ZeroMQ transport over 10 Gb/s Ethernet, since Flink's default, Kafka, is not

fast enough and it does not ingest data over RDMA. (2) The Sender generates records of numerical values rather than JSON strings. We run Flink on KNL by configuring HBM and DRAM in cache mode, so that Flink transparently uses the hybrid memory. We also compare on the high end Xeon server (X56) from Table 3.4 because Flink targets such systems. We set the same target egress delay (1 second) for both engines.

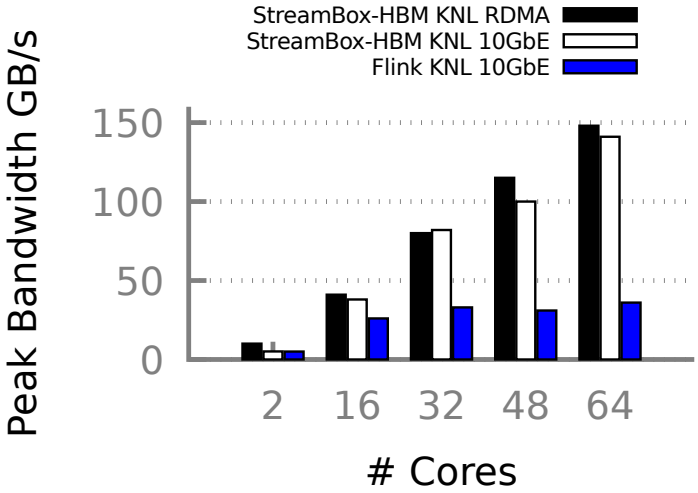
Figure 3.7 shows throughput (a) and peak bandwidth (b) of YSB as a function of hardware parallelism (cores). StreamBox-HBM achieves much higher throughput than Flink on KNL. It also achieves much higher per-dollar throughput on KNL than Flink running on X56, because KNL cost is $5,000, 4.6× lower than X56 at $23,000. Figure 3.7 shows when both engines ingest data over 10 Gb/s Ethernet on KNL, StreamBox-HBM maximizes the I/O throughput with 5 cores while Flink cannot saturate the I/O even with all 64 cores. By comparing these two operating points, StreamBox-HBM shows 18× per core throughput than Flink. On X56, Flink saturates the 10 Gb/s Ethernet I/O when using 32 of 56 cores. As shown in Figure 3.7b, when StreamBox-HBM saturates its ingestion I/O, adding cores will further increase the peak memory bandwidth usage which results from StreamBox-HBM executing grouping computations with higher parallelism. This parallelism does not increase the overall pipeline throughput which is bottlenecked by ingestion, but it reduces the pipeline's latency by closing a window faster. Once we replace StreamBox-HBM's 10 Gb/s Ethernet ingestion with 40 Gb/s RDMA, its throughput further improves by 2.9× (saturating the I/O with 16 cores), leading to 4.1× higher machine throughput than Flink. Overall, StreamBox-HBM achieves 18× higher per core throughput than Flink.

Qualitative comparisons

Other engines, e.g., Spark, and Storm, report lower or comparable performance to Flink, with at most tens of millions of records/sec per machine [2], [3], [31]–[33], [88]. None reports 110 M records/sec on one machine as StreamBox-HBM does (shown below). Executing on a 16-core CPU and a high-end (Quadro K500) GPU, SABER [89] reports 30 M records/sec on a benchmark similar to Windowed Average, which is 4× lower than StreamBox-HBM as shown in Section 3.7.2. On a 24-core Xeon server, which has much higher core frequency



(a) Input throughput under 1-second target delay. Note: X56's 10GbE NIC is slightly faster than that on KNL.



(b) Peak memory bandwidth usage of HBM

Figure3.7. StreamBox-HBM achieves much higher throughput and memory bandwidth usage than Flink, quickly saturating IO hardware. Legend format: "Engine Machine IO". Benchmark: YSB [59]

than KNL, Tersecades [82], a highly optimized version of Trill [12], achieves 49 M records/sec on the same Windowed Average benchmark; compared to it, StreamBox-HBM achieves 2.3× higher machine throughput and 3.5× higher per core throughput before saturating the I/O. In summary, StreamBox-HBM achieves much higher single-node performance than existing streaming engines.

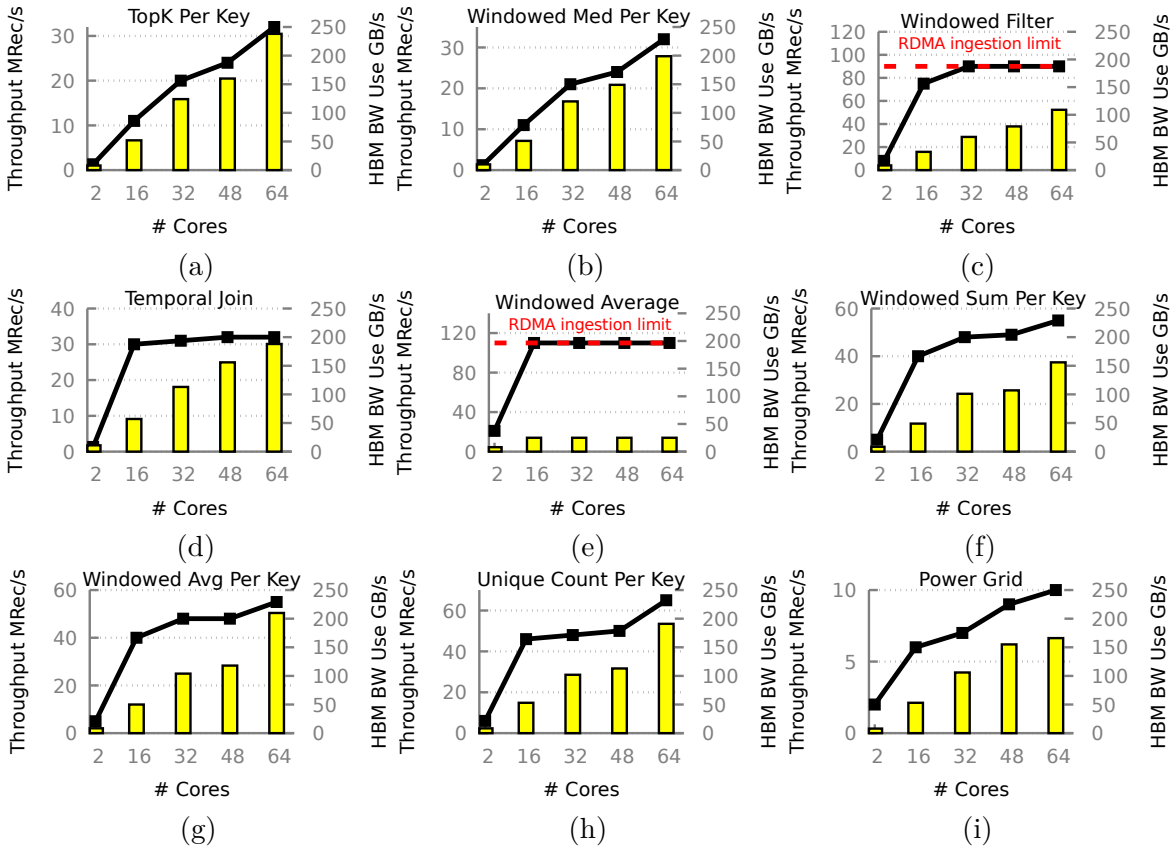


Figure3.8. StreamBox-HBM's throughput (as lines, y-axis on left) and peak bandwidth utilization of HBM (as columns, y-axis on right) under 1-second target output delay. StreamBox-HBM shows good throughput and high memory bandwidth usage

3.7.2 Throughput and Bandwidth

We use nine benchmarks and experimental setup described in Section 3.6 to demonstrate that StreamBox-HBM: (1) supports simple and complex pipelines, (2) well utilizes HBM bandwidth, and (3) scales well for most pipelines.

Throughput and scalability

Figure 3.8 shows throughput on the left y-axis as a function of hardware parallelism (cores) on the x-axis. StreamBox-HBM delivers high throughput and processes between 10 to 110 M records/s while keeping output delay under the 1-second target delay. Six benchmarks scale well with hardware parallelism and three benchmarks achieve their maximum through-

put at 16 or 32 cores. Scalability diminishes over 16 cores in a few benchmarks because the engine saturates RDMA ingestion (marked as red horizontal lines in the figures). Most other benchmarks range between 10 and 60 M records/sec. The simple Windowed Average pipeline achieves 110 M records/sec (2.6 GB/s) with 16 participating cores. StreamBox-HBM's good performance is due its effective use of HBM and its creation and management of parallelism.

Memory bandwidth utilization

StreamBox-HBM generally utilizes HBM bandwidth well. When all 64 cores participate, most benchmarks consume 150–250 GB/sec, which is 40%–70% of the HBM bandwidth limit. Furthermore, the throughput of most benchmarks benefits from this bandwidth, which far exceeds the machine's DRAM peak bandwidth (80 GB/sec). Profiling shows that bandwidth is primarily consumed by Sort and Merge primitives for data grouping. A few benchmarks show modest memory bandwidth use, because their computations are simple and their pipeline are bound by the IO throughput of ingestion.

3.7.3 Demonstration of Key Design Features

This section compares software and hardware StreamBox-HBM configurations, demonstrating their performance contributions.

HBM hardware benefits

To show HBM benefits versus other changes, we configure our system to use only DRAM (StreamBox-HBM DRAM) and compare to StreamBox-HBM in Figure 3.9. StreamBox-HBM DRAM reduces throughput by 47% versus StreamBox-HBM. Profiling reveals performance is capped due to saturated DRAM bandwidth.

Efficacy of KPA

We demonstrate the extraction benefits of KPA on HBM by modifying the engine to operate on full records. Because HBM cannot hold all streaming data, we use cache mode,

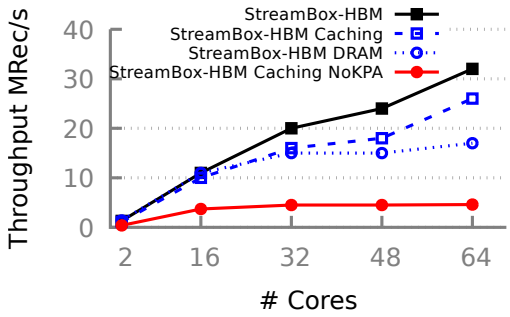


Figure3.9. StreamBox-HBM outperforms alternative implementations, showing the efficacy of KPA and its management of hybrid memory. Benchmark: TopK Per Key

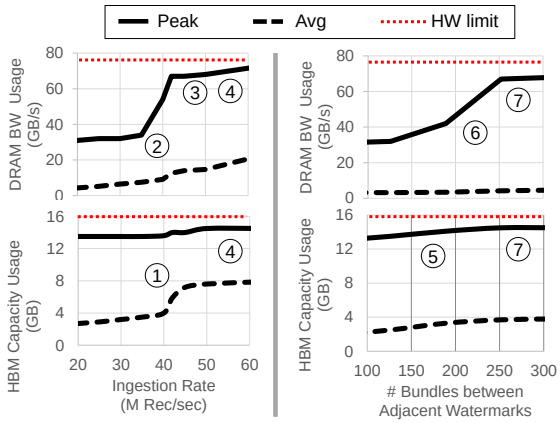


Figure3.10. StreamBox-HBM dynamically balances its demands for limited memory resources under varying workloads. Benchmark: TopK Per Key

thus relying on the hardware to migrate the data between HBM and DRAM (StreamBox-HBM Caching NoKPA). This configuration still uses sequential-access computations, just not on extracted KPA records. It is StreamBox [2] with sequential algorithms on hardware-managed hybrid memory. Figure 3.9 shows StreamBox-HBM outperforms StreamBox-HBM Caching NoKPA consistently on all core counts by up to 7×. Without KPA and software management of HBM, scaling is limited to 32 cores. The performance bottleneck is excessive data movement due to migration and grouping full records.

Explicit KPA placement

StreamBox-HBM fully controls KPA placement and eschews transparent management by the OS or hardware. To show this benefit, we run KPA by turning off KPA placement and configuring HBM and DRAM in cache mode (StreamBox-HBM Caching). This configuration still enjoys the KPA mechanisms, but relies on hardware caching to migrate KPAs between DRAM and HBM. Figure 3.9 shows StreamBox-HBM Caching drops throughput up to 23% compared to StreamBox-HBM. The performance loss is due to excessive copying. All KPAs must be first instantiated in DRAM before moving to HBM. The hardware may move full records to HBM, paying a cost while having little performance return. For stream processing, software manges hybrid memories better than hardware.

Balancing memory demands

To show how StreamBox-HBM balances hybrid memory demands dynamically, we increase data ingress rates to increase memory usage. Figure 3.10a shows when we increase the ingestion rate, HBM capacity usage increases ①. StreamBox-HBM kicks in to counterbalance the trend, allocating more KPAs on DRAM ②. Computation on the extra KPAs on DRAM substantially increases DRAM bandwidth utilization. StreamBox-HBM controls the peak value at 70 GB/sec, close to the DRAM bandwidth limit without saturating it ③. As ingestion rate increases, StreamBox-HBM keeps both resources highly utilized without exhausting them by adding back pressure to ingestion ④. Figure 3.10b shows when we delay ingestion watermarks, which extends KPA lifespans in HBM, adding pressure on HBM capacity ⑤. Observing the increased pressure, StreamBox-HBM allocates more KPAs on DRAM, which increases DRAM bandwidth usage ⑥. As pressure on both resources increases, StreamBox-HBM keeps utilization of both high without exhausting them ⑦.

3.7.4 Impact of Data Parsing at Ingestion

Our design and evaluation so far focus on a common situation where the engine ingests and processes numerical data [82]. Yet, some streaming systems may ingest encoded data,

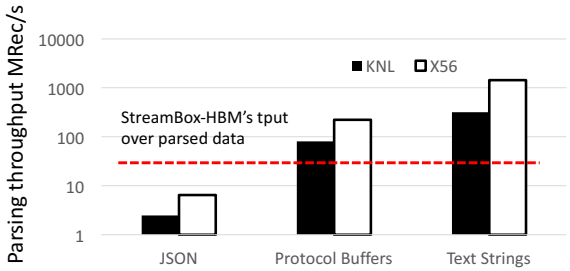


Figure3.11. Parsing at the ingestion shows varying impacts on the system throughput. All cores on KNL and X56 are in use. Parsers: RapidJSON [90], Protocol Buffers (v3.6.0) [91], and text strings to uint64 [92]. Benchmark: YSB

parsing the data before processing. To examine how data parsing would impact StreamBox-HBM's throughput, we construct microbenchmarks that parse the encoded input for the YSB benchmark. We tested three popular encoding formats: JSON, Google's Protocol Buffers, and simple text strings. We run these microbenchmarks on KNL and X56 (listed in Table 3.4) to see if the parsing throughputs can keep up with StreamBox-HBM's throughput on YSB.

As shown in Figure 3.11, parsing at the ingestion shows varying impacts, depending on the ingested data format. While parsing simple text strings can be 29× as fast as StreamBox-HBM processing the parsed numerical data, parsing protocol buffers is 4.4× as fast, and parsing JSON is only 0.13× as fast. Our results also show that data parsing on X56 is 3-4× faster than KNL in general.

Our results therefore have two implications towards fast stream processing when ingested data must be parsed first. First, one shall consider avoiding ingested data formats (e.g. JSON) that favor human-readability over efficient parsing. Data in such formats shall be transcoded near the data sources. Second, since KNL excels at processing numerical data but is disadvantaged in data parsing, system administrators may team up Xeon and KNL machines as a hybrid cluster: the Xeon machines parse ingested data and the KNL machines run StreamBox-HBM to execute the subsequent streaming pipeline.

3.8 Related Work

Stream analytics engines

Much prior work improves stream analytics performance on a single node. StreamBox coordinates task and data parallelism with a novel out-of-order bundle processing approach, achieving high throughput and low latency on multicores [2]. SABER accelerates streaming operators using multicore CPU and GPU [89]. Other work uses FPGA for stream processing [93]. No prior work, however, optimizes stream analytics for hybrid memories. StreamBox-HBM complements prior work that addresses diverse needs in distributed stream processing [3], [8], [9], [13], [44], [88]. They address issues such as fault tolerance [3], [8], [9], programming models [13], and adaptability [43], [94]. As high throughput is fundamental to distributed processing, StreamBox-HBM can potentially benefit those systems regardless of their query distribution methods among nodes.

Managing keys and values

KPA is inspired by key/value separation [66]. Many relational databases store records in columnar format [63], [64], [67], [68] or use an in-memory index [65] to improve data locality and speed up query execution. For instance, Trill applies columnar format to bundles to efficient process only accessed columns, but extracts all of them at once [12]. Most prior work targets batch processing and therefore extracts columns ahead of time. By contrast, StreamBox-HBM creates KPAs dynamically and selectively – only for columns used to group keys. It swaps keys as needed, maintaining only one key from a record in HBM at time to minimize the HBM footprint. Furthermore, StreamBox-HBM dynamically places KPAs in HBM and DRAM based on resource usage.

Data processing for high memory bandwidth

X-Stream accelerates graph processing with sequential access [41]. Recent work optimized quick sort [95], hash joins [75], scientific workloads [53], [54], and machine learning [55] for KNL's HBM, but not streaming analytics. Beyond KNL, Mondrian [69] uses hardware

support for analytics on high memory bandwidth in near-memory processing. Together, these results highlight the significance of sequential access and vectorized algorithms, affirming StreamBox-HBM's design.

Managing hybrid memory or storage

Many generic systems manage hybrid memory and storage. X-mem automatically places application data based on application execution patterns [96]. Thermostat transparently migrates memory pages between DRAM and NVM while considering page granularity and performance [97]. CoMerge makes concurrent applications share heterogeneous memory tiers based on their potential benefit from fast memory tiers [98]. Tools such as ProfDP measure performance sensitivity of data to memory location and accordingly assist programmers in data placement [99]. Unlike these systems that seek to make hybrid memories transparent to applications, StreamBox-HBM constructs KPAs specifically for HBM and fully controls data placement for stream analytics workloads. Several projects construct analytics and storage software for hybrid memory/storage [100], [101]. Most of them target DRAM with NVM or SSD with HDD, where high-bandwidth memory/storage delivers lower latency as well. Because HBM lacks a latency advantage, borrowing from these designs is not appropriate.

3.9 Summary

We present StreamBox-HBM, a stream analytics engine that transforms streaming data and computations to exploit hybrid HBM and DRAM memory systems. It performs sequential data grouping computations primarily in HBM. StreamBox-HBM dynamically extracts into HBM one set of keys at a time together with pointers to complete records in a data structure we call *Key Pointer Array* (KPA), minimizing the use of precious HBM memory capacity. To produce sequential accesses, we implement grouping computations as sequential-access parallel sort, merge, and join with wide vector instructions on KPAs in a streaming algorithm library. These algorithms are best for HBM and differ from hash-based grouping on DRAM in other engines [2], [3], [7], [56], [57].

StreamBox-HBM dynamically manages applications' streaming pipelines. At ingress, StreamBox-HBM allocates records in DRAM. For grouping computations for key k, it dynamically allocates extracted KPA records for k on HBM. For other streaming computations such as reduction, StreamBox-HBM allocates and operates on *bundles* of complete records stored in DRAM. Based on windows of records specified by the pipeline, the StreamBox-HBM runtime further divides each window into bundles to expose data parallelism in bottleneck stream operations. It uses bundles as the unit of computation, assigning records to bundles and threads to bundles or KPA. It detects bottlenecks and dynamically uses out-of-order data and pipeline parallelism to optimize throughput and latency by producing sufficient software parallelism to match hardware capabilities.

The StreamBox-HBM runtime monitors HBM capacity and DRAM bandwidth (the two major resource constraints of hybrid memory) and optimizes their use to improve performance. It prevents either resource from becoming a bottleneck with a single control knob: a decision on where to allocate new KPAs. By default StreamBox-HBM allocates KPAs on HBM. When the HBM capacity runs low, StreamBox-HBM gradually increases the fraction of new KPAs it allocates on DRAM, adding pressure to the DRAM bandwidth but without saturating it.

We evaluate StreamBox-HBM on a 64-core Intel Knights Landing with 3D-stacked HBM and DDR4 DRAM [58] and a 40 Gb/s Infiniband with RDMA for data ingress. On 10 benchmarks, StreamBox-HBM achieves throughput up to 110 M records/s (2.6 GB/s) with an output delay under 1 second. We compare StreamBox-HBM to Flink [57] on the popular YSB benchmark [59] where StreamBox-HBM achieves 18× higher throughput per core. Much prior work reports results without data ingress [2], [12]. As far as we know, StreamBox-HBM achieves the best reported records per second for streaming *with ingress* on a single machine.

The key contributions are as follows. (1) New empirical results find on real hardware that sequential sorting algorithms for grouping are best for HBM, in contrast to DRAM, where random hashing algorithms are best [60]–[62]. Based on this finding, we optimize grouping computations with sequential algorithms. (2) A dynamic optimization for limited HBM capacity that reduces records to keys and pointers residing in HBM. Although key/value separation is not new [12], [63]–[68], mostly it occurs statically ahead of time, instead of

selectively and dynamically. (3) Our novel runtime manages parallelism and KPA placement based on both HBM's high bandwidth and limited capacity, and DRAM's high capacity and limited bandwidth. The resulting engine achieves high throughput, scalability, and bandwidth on hybrid memories. Beyond stream analytics, StreamBox-HBM's techniques should improve a range of data processing syshbmtems, e.g., batch analytics and key-value stores, on HBM and near-memory architectures [69]. To our knowledge, StreamBox-HBM is the first stream engine for hybrid memory syshbmtems. The full source code of StreamBox-HBM is available at http://xsel.rocks/p/streambox.

4. SWAPNN: TOWARDS OUT-OF-CORE NEURAL NETWORKS ON TINY MICROCONTROLLERS

4.1 Introduction

With low cost and energy, MCUs are becoming ubiquitous platforms for neural networks (NNs), a paradigm dubbed tinyML [102]. Running NN *on MCU*, rather than sending raw data off, offers multiple advantages, notably tolerating poor networks and preserving data privacy. Use cases include detecting farming crop disease by classifying leaf photos [103] and extracting traffic patterns by analyzing city images.

A top obstacle in tinyML is memory limit. On one hand, an MCU has small memory, which comprises tens to hundreds KB of SRAM as the main memory and byte-addressable flash of no more than a few MBs for read-only data. Note that the byte-addressable flash is different from external block-addressable storage such as SD cards [104].

On the other hand, state-of-the-art NNs achieve high accuracy and generality with large memory footprints [105], [106]. An NN's memory footprint includes read-only parameters and intermediate/final results called feature maps. Although MCU can process one NN layer in memory before loading the next layer, a layer's parameters and feature maps can still take up to 100 MB (e.g. VGG16 [107]). This exceeds the MCU memory size by up to two orders of magnitude. Such a memory gap is widening as recent NNs are becoming larger [108] while MCU memory sees slow, if at all, scaling due to cost constraints [109].

A popular approach to overcoming memory limitation is to engineer NNs themselves. Common techniques include model compression [111]–[113], parameter quantization [114], designing tiny NNs from scratch [115], as well as automation of these procedures [116]. In exchange, this approach gives away model accuracy or generality at varying degrees. Unfortunately, in order for an NN to fit into the MCU memory, the NN either becomes substantially inaccurate (e.g. $< 60\%$ top-1 accuracy as shown in Figure 4.1) or too specialized (e.g. can only detect a few object classes [117]).

This disqualifies MCUs from the use cases where high accuracy/generality are desired while delays can be tolerated, for example: (1) *NN inference on slowly changing signals*, e.g., monitoring crop health by analyzing hourly photos [103] and traffic patterns by analyzing

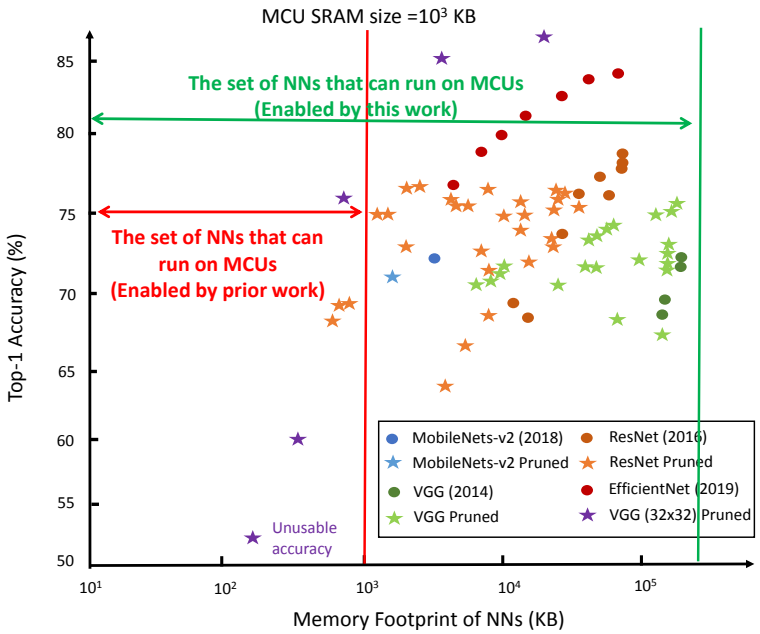


Figure4.1. Many popular NNs exceed the MCU memory size [110].

video frames every 20-30 minutes [117]. (2) *profiling NNs on device*: occasionally running a full-blown NN to estimate the accuracy of long-running smaller NNs [118]; (3) *transfer learning*: re-training NNs on MCUs with data collected from deployment every hour or day [119].

A case for out-of-core NNs

Can an MCU execute NNs that far exceed its physical memory size? A proven wisdom is to dynamically swap *tiles* of NN layers between memory tiers [120]. Specially, an MCU runtime can split one NN layer's working set into a series of tiles, each small enough to fit the MCU memory; load tiles from external storage (a micro SD card) to memory, compute on them, and write results back to the storage for subsequent processing. While prior systems have swapped NN tiles between a server's CPU/GPU memories [121], applying the idea to MCU, in particular swapping between small SRAM and a wimpy SD card, raises multiple concerns: loss of SD card durability, execution slowdown due to IO operations, energy increase, and safety/security of out-of-core NN data. This chapter aims to address these concerns.

Key observations

This chapter demonstrates the practicality of out-of-core NN on MCUs, for which we have following observations.

- *Swapping overhead is only pronounced in certain NN layers.* Only on layers with low arithmetic intensity, notably fully connected (FC) layers, the swapping delay due to IO is longer than that of computation; on layers with higher arithmetic intensity, e.g. convolution (Conv), the swapping delay is dwarfed by that of computation. The swapping overhead is further diminished by MCU's relative low CPU speed as compared to its IO speed.

- *Swapping rate is throttled by computation*, which limits the wear rate of SD cards. As a common NN structure, IO-bound layers such as FC are spaced by compute-bound layers such as Conv. As a result, even with continuous NN executions, IO is only exercised intermittently.

- *Most IO traffic for swapping is read* This is because a layer's parameters and input feature maps are often much larger than its output feature maps. Fortunately, read traffic does not wear SD cards.

- *Hide swapping delays with parallelism at various granularities.* Within a layer, the MCU can exploit *tile* parallelism, by computing on a tile while transferring others to/from the storage. Between consecutive NN executions such as on a sequence of video frames, the MCU can further exploit *pipeline* parallelism, by overlapping the swapping IO for an earlier frame with the computation of a later frame.

- *Modern MCU hardware often over-provision durability.* For example, a 64 GB SD card can last more than 10 years with 100 GB of daily writes (Section 4.5.4). As such, MCU can trades the surplus durability as a system resource for accommodating large NNs. Modern MCUs incorporate rich specialized hardware, e.g., for DMA, hash, and crypto, which accelerates and secures IO operations.

- *IO adds marginal energy to an already busy MCU.* With an MCU already busy on computation, most of its hardware components in high power states. Further activating the SD card increases the system energy moderately.

Quantitative findings

We present SwapNN, a scheduler design that automatically schedules IO and compute tasks. SwapNN exploits the IO/compute parallelism across tiles, layers, and data frames, meanwhile respects memory constraint and data dependency. We applied SwapNN to a diverse set of NNs, MobileNets [122], AlexNet [123], and VGG16 [107], on a Cortex-M7 MCU with 340 KB of SRAM. Our findings are:

- *Low to modest speed overhead.* NNs with dominant compute-bound layers see negligible swapping overhead, both in per-frame delay and frame throughput. Compared to running VGG on an *ideal* MCU with infinite main memory (SRAM), out-of-core execution with 512 KB memory sees only 6.9% longer per-frame delay and only 3% lower throughput. 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.

- *Large tiles are crucial to low swapping overhead.* A key parameter in out-of-core NN is the tile size, which determines the granularity of IO/compute task. While small tiles lead to fine-grained tasks and therefore better compute/IO parallelism, they increase the total amount of IO traffic and the per-byte IO delay. As we will show experimentally, the cost of small tiles overshadows the benefit of parallelism on typical MCU hardware and NNs,

- *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, comparable to SD card writes on a commodity surveillance camera. A 64 GB SD card can sustain such a write rate for 7.5 years before half of its cells are worn out.

- *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).

- *Out-of-core data can be secured* with known mechanisms, such as encryption and hash-based integrity protection. Specialized hardware on MCUs further reduces their overhead.

Contributions

Our contributions are as follows.

- We present the first study of applying swapping to NN on MCUs. We analyze the swapping-generated IO activities and their implications on performance, storage durability, energy, and data security.

- We explore software/hardware parameters that impact swapping overhead. Towards lowering swapping overhead, our findings shed light on setting software parameters and designing MCU hardware (e.g., choosing SRAM size).

- We present a scheduler design that can automatically schedule IO and compute tasks in parallel. The scheduler exploits a common NN characteristic that an NN often has a mix of IO-bound and compute-bound layers. It exploits IO/compute parallelism across NN layers and across data frames while respecting memory constraint and data dependency.

- We make a case that an MCU of less than ten dollars with hundreds of KB SRAM can execute large NNs such as VGG16, which expands the scope of tinyML significantly.

Table4.1. Normalized arithmetic intensity (N) on NN layers with MCU's common speed range (64–480 MOPS [124], [125]) and IO bandwidth range (10–40 MB/s [126]). NN: VGG16

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**

4.2 Background and Motivations

4.2.1 A Taxonomy of NN layers

To study the swapping overhead, we focus on a layer's swapping delay *relative* to its computation delay on typical MCUs. The rationale is that as MCU can perform swapping and computation in parallel, the longer of the two delays will be the layer's bottleneck.

Study setup

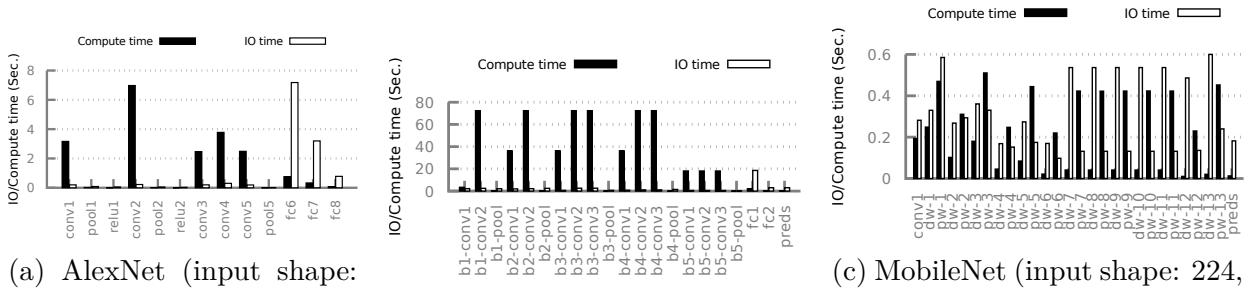
Since the working set of a layer may not fit into SRAM, we split a layer's input, weight parameter, and output into small tiles (e.g., 128KB). For compute time, we measure the time to calculate every output tile, then calculate the layer's compute time by adding every output tile's compute time. For IO time, we measure the time to read input tiles, weight tiles (once), and output tiles, and then calculate the layer's IO time by adding them together. Figure 4.2 shows the IO time and compute time of each layer in three typical CNNs, where the buffer size for tiles is 128 KB.

Classifying NN layers

In general, *arithmetic intensity*, as commonly used in HPC [127], characterizes a workload's compute/IO ratio. It is defined as W/Q, where Q is the amount of data to move in the memory hierarchy and W is the amount of arithmetic operations on the data. By factoring

Table4.2. Number of IO-bound and compute-bound layers and quantized memory footprints of popular NNs [128].

	MobileNets	AlexNet	VGG16	ResNet18	GoogLeNet
Num. of compute-bound layers	14	5	13	16	21
Num. of IO-bound layers	13	3	2	2	6
Size of feature maps (MB)	10	1	15	5	6.5
Size of weight (MB)	4	62	138	11	13
Memory footprint (MB)	14	63	153	16	19.5



(a) AlexNet (input shape: 227)

(b) VGG16 (input shape: 224)

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

Figure4.2. Per-layer compute and IO delays in NNs. (1) Observation: NNs have a mix of IO-bound and compute-bound layers. (2) Insight: IO time can be hidden by compute time with parallel execution. (3) Configuration: MCU is ARM Cortex-M7 @ 216 MHz, tile/buffer size is 128 KB, Transcend SD card size is 32 GB.

in an MCU's CPU speed (S_{cpu}) and IO bandwidth (S_{IO}), we define $N = (W/S_{CPU})/(Q/S_{IO})$ as the *normalized* arithmetic intensity on MCU. Of a given layer, $N > 1$ means swapping incurs less delay than computation, i.e, a compute-bound layer; $N < 1$ means swapping incurs longer delay, i.e. an IO-bound layer.

On modern MCUs with simple CPU cores, S_{CPU} is primiarly determined by the CPU clockrate; it ranges from 64 MOPS to 480 MOPS [124], [125]. S_{IO} is jointly determined by the MCU's DMA bandwidth and the SD card bandwidth, ranging from 10 MB/s to 40 MB/s as reported in the literature [126]. With these values, common NN layers fall into three distinct categories per their normalized arithemetic intensity (N).

(1) A majority of compute-bound layers ($N >> 1$). Notable examples are Conv layers known for their high complexity. In the example of VGG16 (Table 4.1), N for the Conv layers far exceeds 1 even with a high CPU clockrate and slow IO. They often dominate an NN's execution time (51% – 90%), as exemplified by the three NNs in Figure 4.2. On these layers, the computation delay overshadows the IO delay.

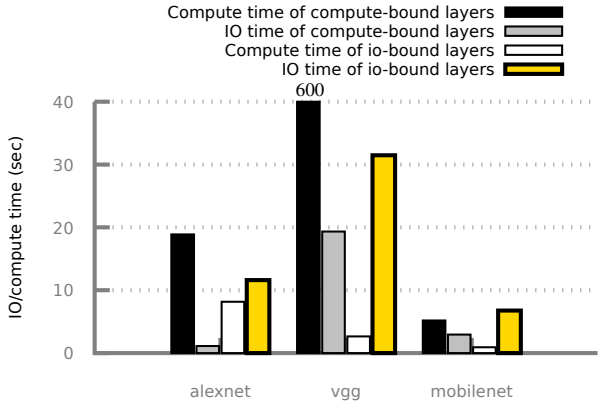


Figure4.3. IO/compute delays in out-of-core NN execution. The total execution delay is dominated by compute in the compute-bound layers and IO in the IO-bound layers.

(2) Some IO-bound layers ($N < 1$). Examples include fully connected (FC) and depth-wise convolutional layers (DW). These layers perform light computation over large volumes of feature maps and weight parameters. Of all layers in an NN, they are often minorities (e.g. 2 out of 21 in VGG16). With out-of-core execution, the IO delay exceeds the computation delay by up to 10× (e.g. fc1 in Table 4.1 and Figure 4.2b).

(3) Other layers with insignificant overheads, e.g., Relu and Maxpooling. These layers have low complexity and contribute a tiny fraction of data to move and to compute (0.3%-0.9%) for an NN. As such, their swapping overhead is insignificant.

Common pattern of NN layers

Based on the NN layer classification, there are two common patterns in typical CNNs:

(1) CNNs have a mix of compute-bound and IO-bound layers, and the number of compute-bound layers is usually larger than other layers. Table 4.2 shows the number of compute-bound and IO-bound layers in typical CNNs. For instance, MobileNets [122], Alexnet [123], VGG16 [107], ResNet18 [129], and GoogLeNet [105] have 14/13, 5/3, 13/2, 16/2, and 21/6 of compute-bound/IO-bound layers respectively.

(2) The overall CNN execution time is dominated by the compute time of compute-bound layers and the IO time of IO-bound layers. Figure 4.3 shows the IO time and compute

of IO-bound/compute-bound layers. For instance, compute time of compute-bound layers dominate the overall time in Alexnet and VGG. For Mobilenet, the IO-time of IO-bound layers dominates the overall time, because Mobilenet is using specially point-wise and depth-wise convolutions [122], which have lower compute complexity than general convolutional layers.

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

4.2.2 The System Model

MCU hardware

We assume the following hardware components: (1) a CPU with clockrate from tens of MHz to a few hundred MHz, as exemplified by Arm Cortex M3 and M7; (2) on-chip SRAM: from tens of KBs to several MBs; (3) on-chip NOR flash: byte-addressable, read-only memory no more than a few MBs; (4) cheap external storage, e.g. a micro SD card ranging from tens of GBs to a few hundred GBs; (5) a DMA engine, for moving data between SRAM and external storage without CPU involved; (6) optionally, on-chip accelerators for computing crypto and hash functions.

Major vendors ship numerous MCU models meeting the above conditions. Examples include the STM32 MCU family from STMicroelectronics [130] and the LCP series from NXP Semiconductors [131]. They are priced at $1-$20 per unit.

NN workloads & metrics

We motivate our study by considering periodic NN inference on video/audio data as a sequence of *frames* captured by MCUs at run time. To characterize inference speed, we consider both the inference delay of each frame and throughput as the number of frames processed per second. MCU applications may be sensitive to either metric or both. For instances, keyword spotting is sensitive to inference delays [132] and car counting benefits from high throughput [117].

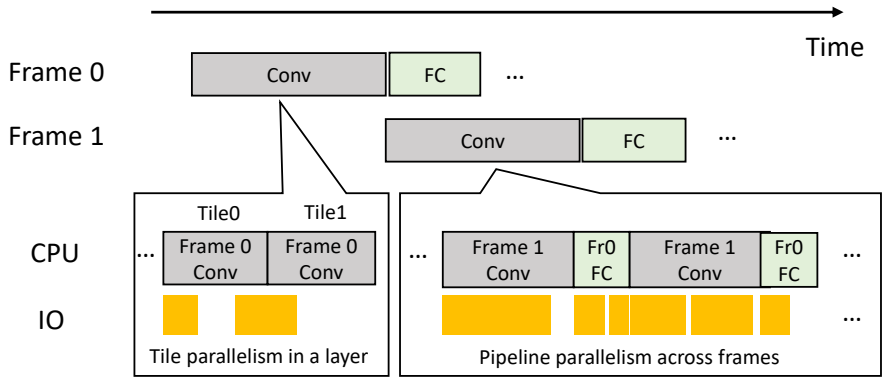


Figure4.4. An example of out-of-core NN execution, showing Conv (compute-bound) and FC (IO-bound) layers. Note: gray/green boxes show the computation of NN layers in NN layers/frames, and yellow boxes show the IO operation in NN layers/frames.

Out-of-core NN executions

We consider the following swapping strategy. An NN's parameters are pre-stored on the external flash. Given an input frame, the MCU executes the NN's layers in sequence. It processes a layer in tiles, in case the layer's memory footprint exceeds MCU's main memory: to do so, the MCU loads to the main memory a tile of parameters and a tile of input feature maps, computes a tile of output feature maps in memory, and writes back the output to the external flash. Altogether, the input and output tiles shall simultaneously fit in the main memory.

As shown in Figure 4.4, MCU extracts CPU/IO parallelism for hiding IO delays. (1) *Tile parallelism within an NN layer*: while computing an output tile *Tile0*, MCU can preload from flash the input tiles for computing the next output tile *Tile1*; while writing back the completed *Tile0* back to flash, MCU can compute *Tile1* simultaneously. (2) *Layer parallelism*: in a similar fashion, MCU can execute an earlier layer's computation with a latter layer's IO simultaneously. (3) *Pipeline parallelism across data 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. As shown in Figure 4.4, MCU swaps frame 0's FC layer while computes on frame 1's Conv layer.

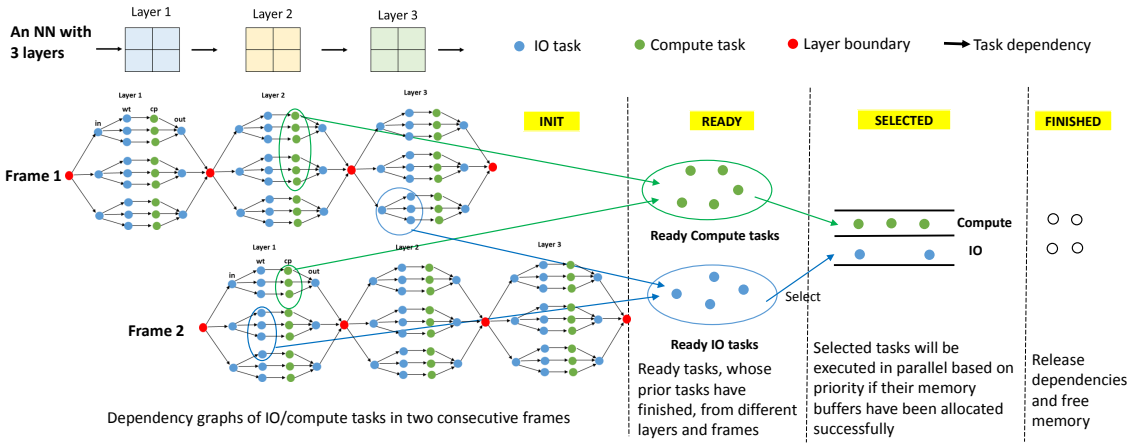


Figure4.5. Overview of SwapNN: scheduling IO/compute tasks across tiles, layers, and frames in parallel according to dependencies, priorities, and memory constraints.

4.3 SwapNN: Automatically Scheduling IO/Compute Tasks in Parallel

In order to reduce IO overhead in swapping, we present SwapNN, a scheduler design that automatically schedules IO tasks and compute tasks across tiles, layers and frames in parallel based on NN characteristics, meanwhile respects memory constraint and data dependency.

4.3.1 Challenges

As shown in Figure 4.4, MCU ideally could extract CPU/IO parallelism for hiding IO delays. However, such ideal parallel scheduling sequence is difficult to find because it must meet the following requirements at the same time: (1) the scheduler must automatically identify what tiles should be executed in parallel according to their dependencies and relative IO/compute time; (2) the working set of tiles being executed in parallel must be smaller than SRAM at every single moment; (3) the parallel sequence should keep both MCU core and IO bandwidth fully utilized to avoid either of them from idling.

Furthermore, divers NN layers with different parameters and diverse SRAM sizes on MCUs create a huge space of choices for deciding parallel sequence for IO/compute tasks, which makes parallel scheduling even more difficult.

4.3.2 SwapNN Design

To address the above challenges, we present the design of SwapNN, describing how to decide tile size, manage memory buffers, and schedule IO/compute tasks in parallel, meanwhile respect memory constraint, data dependency, and task priority.

Tiling NN layers and managing memory buffers

A key question in swapping is to decide tile sizes for NN layers based on SRAM size. SwapNN splits SRAM size into fixed number of buffers, and then calculates tile sizes based on layer parameters and buffer size, which are totally transparent to users' application code. Specifically, the input tile size depends on the output tile size, so they will be decided together and the larger one of them must be smaller than buffer size. Weight tile size doesn't depend on input or output, so it is calculated just according to weight size and buffer size.

As show in Algorithm 1& 2, SwapNN equally splits SRAM into buffers with fixed size, and creates three separate memory buffer pools for input feature maps, weight parameters, and output feature maps, who have 1/4, 1/2, and 1/4 of total memory buffers. The reason why SwapNN creates separate memory pools, instead of one pool, is that single memory pool for input/weight/output tiles leads to deadlock in parallel execution. For example, all memory buffers may be allocated to input and weight tiles, so execution cannot continue because of no memory buffers for output tiles. The rational to choose 1/4, 1/2, and 1/4 is based on the minimal parallel working set of computing one output tile, which includes one input tile, at least two weight tiles, and one output tile.

NN task and graph

As shown in Figure 4.5, SwapNN defines two types of tasks: IO task and compute task. An IO task reads/writes tiles from/to SD, and a compute task computes an output tile based on corresponding input/weight tiles.

SwapNN defines an NN as a computation graph $G = (V, E)$, where V is the node set of IO and compute tasks, and E is the edge set representing dependencies. For instance, a

compute task depends on IO tasks that read input/weight tiles, and a write IO task depends on a compute task that finishes computing output tile. Every task has a set of properties, e.g., *in-degree* counter indicating the number predecessors of current task, memory buffer and tile sizes, execution time, and execution priority.

Two things that are worth noting in NN graph: (1) we enforce dependencies between an input tile and multiple weight tiles to ensuring reading input tile first, so that reading other weight tiles can happen in parallel with computing an output tile. (2) each output tile depends on all weight tiles, so weight tiles may be read multiple times (once for each output tile) during execution.

As show in Algorithm 1& 2, *BuildGraph()* takes NN architecture and SRAM size as parameters. For each layer, SwapNN: (1) calculates tile sizes for input/weight/output based on memory buffer size; (2) creates read IO tasks for input and weight tiles, compute tasks for computing output tiles, and write IO tasks for output tiles; (3) inserts IO/compute tasks to execution graph based on dependencies; (4) sets task properties, including execution time, memory buffer size, *inDegree* counter, and priority.

Task state

As shown in Figure 4.5, SwapNN defines the following states for every IO/compute task to manage their lifecycle:

- **INIT** A task is set to INIT state when building the execution graph based on NN architecture, layer parameters, SRAM size, buffer size, and dependency.
- **READY** A task becomes READY when all of its predecessors have finished, at which point the *in-degree counter* of the task drops to zero.
- **SELECTED** A task switches to SELECTECD from READY when its memory buffers has been successfully allocated, e.g., an input/weight buffer for a read IO task or an output buffer for a compute task.

- **FINISHED** When a IO/compute task is finished, it switches to FINISHED state, at which point SwapNN decreases the *in-degree counter* by one for the task's all successors to release the dependency and free memory buffers accordingly.

Task priority

When there are multiple READY tasks from multiple layers and frames, the tasks from earlier frames/layers should have higher priority to be executed to guarantee per frame delay. SwapNN assigns priority to tasks based on their frame number and layer number when creating these tasks, and schedules them at runtime according to the priority.

Scheduling NN tasks

Given tiling strategies of NN layers, SwapNN finds the optimal parallel sequences for IO and compute tasks based on their dependencies, available memory buffers, and priority. One goal of scheduling is to keep both MCU and IO busy to avoid either of them from idling, to achieve low latency and high throughput.

As show in Algorithm 1&2, SwapNN maintains two tasks queues, *ReadyIO* and *ReadyCP*, for READY IO tasks and READY compute tasks respectively. READY tasks in these two queues are sorted based on their priority, and the one with the highest priority will be scheduled each time.

ScheduleIOTask() keeps looking for IO tasks in ReadyIO queue in priority order. For write IO tasks that do not require memory allocation, SwapNN issues write DMA operation, and then frees memory buffers and releases the dependencies for the task's successors. For read IO task, SwapNN first tries to allocate memory buffer for it. If the allocation succeeds, then issue read DMA operation and release the dependencies for the task's successors.

ScheduleComputeTask() keeps looking for compute tasks in ReadyCP queue in priority order. It first tries to allocate memory buffers to store computing output. If the allocation succeeds, SwapNN executes the compute task, release the dependencies for its successors, and free memory buffers of input/weight tiles.

Algorithm 1: Scheduling IO/compute tasks in parallel

```
Input  : NN architecture and SRAM size
Layers[L] = parameters of L layers in an NN
ReadyIO = a set of READY IO tasks sorted by priority
ReadyCP = a set of READY Compute tasks sorted by priority
Function BuildGraph(Layers, SRAMSize):
    G = empty graph
    for Layer ∈ Layers do
        Calculate tile sizes for input, weight, and output;
        for all input tiles do
            Insert an IO task for reading input tile to G;
            for all weight tiles do
                Insert an IO task for reading weight tile to G;
                Insert a Compute task to G;
            Insert an IO task for writing output tile to G;
    Set the root IO Task to READY and insert into ReadyIO;
    return G;
Function ScheduleIOTask(G):
    while readyIO is NOT empty() do
        for iotask in ReadyIO do // in priority order
            if iotask is WRITE then
                Execute the write iotask;
                ReleaseSuccessors(G, iotask);
                freeMemoy(iotask.buffptr);
            else // iotask is READ
                buffptr = AllocateMemory();
                if buffptr != NULL then
                    Execute the READ IO task;
                    ReleaseSuccessors(G, iotask);

Function ScheduleComputeTask(G):
    while readyCP is NOT empty() do
        for cptask in ReadyCP do // in priority order
            buffptr = AllocateMemory(); // for output tile
            if buffptr != NULL then
                Execute the Compute task;
                ReleaseSuccessors(G, cptask);
                freeMemory(); // for input and weight tiles

Function ReleaseSuccessors(G, task):
    for suctask in task's successors do
        if suctask.inDegree - - == 0 then
            insert suctask to ReadyIO or ReadyCP;
```

Algorithm 2: Scheduling IO/compute tasks in parallel (continued)

```
freeListIn, freeListWt, freeListOut = lists of free memory buffers for input, weight, and
  output tiles
Function AllocateMemory(freeList):
    if freeList is empty then
        return NULL;
    buffptr = select one buffer from freeList;
    return buffptr;
Function FreeMemory(buffptr, freeList):
    insert buffptr to freeList;
```

With two separate threads running *ScheduleIOTask()* and *ScheduleComputeTask()*, SwapNN can schedule any ready IO and compute tasks in parallel across tiles, NN layers, and data frames, meanwhile respects memory constraint, data dependency, and task priority. Therefor, the IO overhead in swapping can be reduced.

4.4 Implementation & Methodology

Implementation

We implement swapping kernels for typical NN layers to compute tiles on MCU atop CMSIS-NN library [133], and currently supported layers include Convolution, ReLu, Pooling, Fully Connected, Depth-wise convolution, and Point-wise convolution. We implement the scheduler in C++, which can run on desktop to find the best parallel scheduling sequence without deploying on MCUs.

Studied NNs

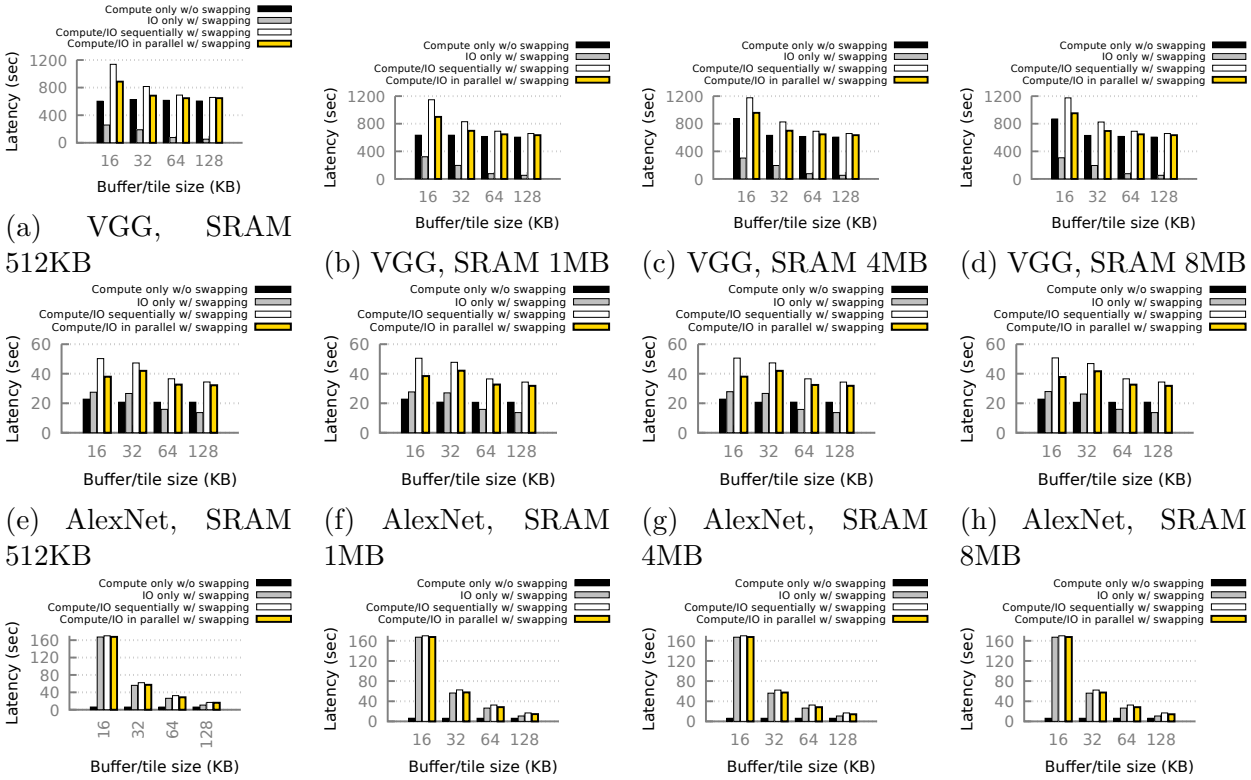
We study three representative NNs, whose memory footprints range from sveral-MB to hundred-MB (with quantization). As shown in Table 4.2: MobileNet has large feature maps but small weight parameters, AlexNet has small feature maps but large weight parameters, and VGG16 has 1000× larger memory footprint than MCUs' SRAM size.

Input data

We use synthetic images as the input. Note that the input contents do not affect NN execution time/efficiency, hence our measurement results.

Methodology

In order to understand how swapping affects the latency, throughput, SD durability, energy consumption, and security, we do the following steps for all three NNs: (1) Given SRAM size and buffer size, calculate the tile sizes for all layers of an NN; (2) Based on tile sizes of layers, we run the swapping kernels as microbenchmarks on target MCU hardware



(a) VGG, SRAM 512KB (b) VGG, SRAM 1MB (c) VGG, SRAM 4MB (d) VGG, SRAM 8MB

(e) AlexNet, SRAM 512KB (f) AlexNet, SRAM 1MB (g) AlexNet, SRAM 4MB (h) AlexNet, SRAM 8MB

(i) Mobilenet, SRAM 512KB (j) Mobilenet, SRAM 1MB (k) Mobilenet, SRAM 4MB (l) Mobilenet, SRAM 8MB

Figure4.6. Swapping latency of NNs with different SRAM sizes and buffer sizes. Observation: swapping incurs negligible or modest delay in latency.

(TM32F746NG-Discovery board: ARM Cortex-M7 at 216 MHz, 340 KB SRAM, 32 GB SD card), and then measure the IO/compute time for tiles; (3) The scheduler takes NN architecture/parameters, SRAM size, buffer/tile sizes, IO/compute time of tiles as parameters, and then automatically finds out the optimal parallel scheduling sequences for IO and compute tasks across layers and frames.

For latency, we measure the time to process one NN frame. For throughput, we measure the time to process 10 consecutive NN frames in parallel and then calculate the throughput. For energy, we measure the *worst-case* energy consumption by keep running IO and compute tasks simultaneously.

4.5 Findings

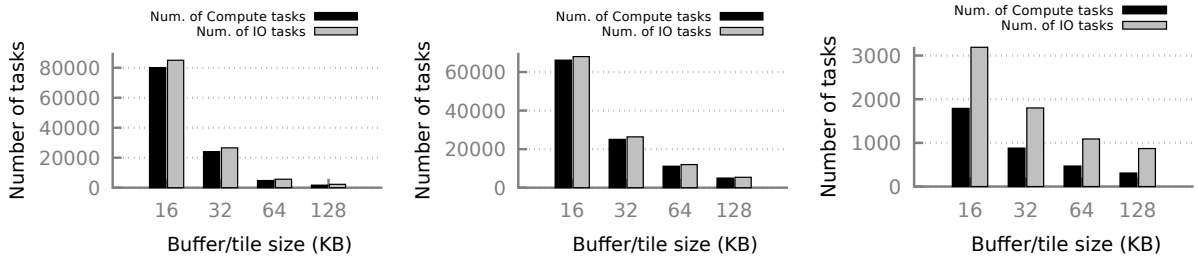
This section focus on the analysis and findings of out-of-core NN on MCUs by answering the following questions:

- What are the parameters and tradeoffs that affect swapping performance?
- How does swapping affect per-frame latency?
- How does swapping affect throughput?
- Will swapping wear out SD soon?
- How much extra energy does swapping consume?
- Does swapping incur security issues?

4.5.1 Software/Hardware Parameters and Their Tradeoffs

There are multiple hardware/software parameters that affect the swapping performance, including SRAM size, buffer/tile size, the number of buffers, NN's memory footprint, and the ratio of compute-bound and IO-bound layers in NNs. We analyze each of them as following:

- *SRAM size:* Large SRAM leads to large memory buffers or more memory buffers, but also increases cost and energy consumption.
- *Buffer/tile size:* Tile is a small chunk of input/weight/output, and it decides the granularity of IO/compute task. Small tiles lead to fine-grained tasks and therefore better compute/IO parallelism, but they increase the total amount of IO traffic/time. Buffers are used to store tiles, and tile size is calculated based on buffer size. We treat them the same in discussion.
- *The number of memory buffers:* The more, the better. More memory buffers allows more tiles co-existing in SRAM, so more tasks can be executed in parallel.
- *NN's memory footprint:* It's decided by NN architecture. NNs with larger memory footprint see higher IO overhead in swapping due to more IO traffic, and vice versa.



(a) VGG (b) AlexNet (c) Mobilenet

Figure4.7. Number of IO/compute tasks in NNs under different buffer/tile sizes. Observation: the number of IO/compute tasks drops significantly as the buffer/tile size increases.

- *The ratio of compute-bound and IO-bound layers in NNs.* It'd decided by NN architecture and affects the IO overhead in swapping. NNs with more compute-bound layers, the IO overhead is lower since IO time can be hidden by relatively longer compute time. In contrast, NNs with more IO-bound layers, the IO overhead is higher since the relatively longer IO time cannot be hidden by compute time.

Tradeoffs in buffer/tile size, the number of buffers, IO traffic/time, and parallelism

Given SRAM size, the buffer/tile size and the number of buffers can be decided, and their tradeoffs effects overall IO traffic/time and parallelism in swapping.

- *Large buffer/tile size leads to low IO traffic/time, but limits execution parallelism*: Given an NN and SRAM size, large buffer/tile size leads to small number of tiles, and hence low IO traffic. The overall IO time is short due to less IO traffic, but the execution parallelism is low due to small number of buffers.

- *Small buffer/tile size leads to high execution parallelism, but increases overall IO traffic/time*: Given an NN and SRAM size, small buffer/tile leads to large number of tiles, and hence high IO traffic. The overall IO time is long due to high IO traffic and more fine-grained IO tasks, but the execution parallelism is high due to large number of memory buffers, which allow more tiles to co-exist in memory and be processed in parallel.

Experimental insights

We study how these parameters affect swapping performance on MCU with experiments, and we have the following findings:

- *Increasing buffer/tile size can significantly reduce the number of IO tasks and overall IO time.*

  The number of IO tasks drops as buffer/tile size increases. Figure 4.7 shows the number of IO/compute tasks of NNs under different buffer sizes. For instance, when buffer/tile size increases from 16 KB to 128 KB, the number of IO tasks (Grey bars in Figure 4.7) of VGG, AlexNet, and MobileNet drops from 85024 to 2248, from 68040 to 5390, and from 3190 to 870 separately.

  Overall IO time drops as buffer/tile size increases. As the IO time (gray bar) shown in Figure 4.6a, Figure 4.6e, and Figure 4.6i, where SRAM size is 512 KB. When buffer/tile size increases from 16 KB to 128 KB, the overall IO time of VGG, AlexNet, and MobileNet drops from 257.824s to 52.4849s, from 27.4765s to 13.6731s, and from 167.183s to 10.6669s separately. The same pattern can also be observed when using larger SRAM sizes in Figure 4.6.

- *Parallel execution can reduce IO overhead, especially when there are larger numbers of buffers.*

  When there are more memory buffers, more IO/compute tasks can be executed in parallel, and hence more IO time can be hidden by compute time. For instance, the white and yellow bars in Figure 4.6a show the sequential execution time and parallel execution time under different buffer sizes (different number of buffers). When buffer/tile size increases from 16 KB to 128 KB, the number of buffers drops from 32 to 4, and IO time reduced by parallel execution drops from 251s to 10s (compared to sequential execution). The same pattern can also be observed in other NNs in Figure 4.6.

- *Given SRAM size, comparing to small buffer/tile size with high parallelism, large buffer/tile size with low parallelism incurs much lower IO overhead in swapping.*

Both large buffer/tile size (small number of buffers) and high parallelism can reduce IO overhead, but they are in conflict and cannot be achieved at the same time. We observe that the former one can reduce more IO time then the later one.

MobileNet is IO-intensive NN, parallel execution cannot reduce IO overhead much even with more buffers (smaller buffer/tile size, e.g, 16 KB). However, increasing buffer size can reduce IO time from 167s to 16s when buffer size increases from 16KB to 128KB, as shown in Figure 4.6i.

The same pattern also can be observed in AlexNet and VGG, but benefit of choosing large buffer/tile size is not as significant as MobileNet because they are less IO-intensive. For these two NNs, parallel execution plays a bigger role to hide IO time when buffer size is small, while low overall IO tasks/time plays a bigger role when buffer size is large. Overall, large buffer/tile size still overshadows the benefit of parallelism.

4.5.2 Impact on Per-frame Delay

Implication: *With large buffer/tile size, NNs with a small fraction of IO-bound layers see negligible delay increase; NNs with more IO-bound layers see modest delay increase.*

Within a compute-bound layer, MCU can execute IO and computation for consecutive tiles simultaneously (as these tiles are independent), completely hiding the IO delay behind the much longer computation delay. Within an IO-bound layer, IO and compute for consecutive tiles can happen simultaneously as well, but the long IO delay cannot be totally hidden by relatively shorter compute delay. For other layers, e.g. relu/pooling, the IO/compute delay is insignificant.

As such, the increased delay of an NN due to swapping is mainly determined by the proportion of IO-bound layers' IO delay to all layers' total compute delay. The increased delay for NNs with less IO-bound layers is negligible. As VGG shown in Table 4.2, only 2 out of 13 layers are IO-bound, leading to only about 6.9% increased delay as shown in Figure 4.6a – Figure 4.6d (Yellow vs. Black bars). The increased delay for NNs with more IO-bound layers is modest. As AlexNet and MobileNet show in Table 4.2, 3 of 5 and 13 of 28 layers are IO-bound, leading to 50% and 150% increased delay when buffer/tile size is as

large as 128 KB, as shown in Figure 4.6e – Figure 4.6l (Yellow vs. Black bars). Overall, the increased delay due to swapping is negligible for compute-intensive NNs and modest for IO-intensive NNs.

***Implication*:**

Insight for hardware designer: increasing SRAM size only increases cost, but cannot improve the latency much in swapping.

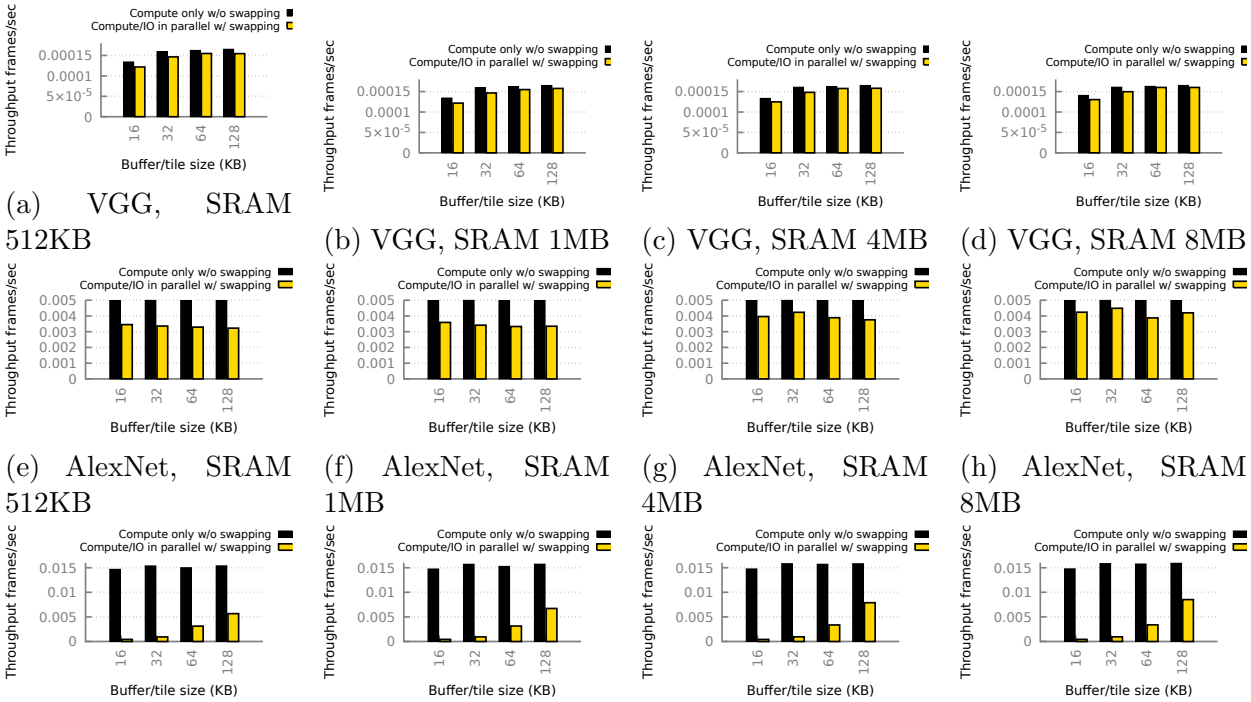
As shown in Figure 4.6, the latency of VGG, AlexNet, and MobileNet does not decrease much as the SRAM size increases. For given buffer size, using larger SRAM can increase the number of buffers, and hence can increase parallelism. However increasing SRAM size and the number of buffers cannot help much, because the gap between the number of tasks and the number of buffer is too large (100× gap). For instance, the number of IO tasks in MobileNet is 55877 (Figure 4.7c) when buffer size is 16 KB, but the number of buffers only increases from 32 to 512 (100× smaller than 55877) when SRAM size increase from 512KB to 8MB.

4.5.3 Impact on NN Throughput

Implication: *With large buffer/tile size, NNs see negligible or modest throughput loss.*

NNs with negligible delay increase will also see negligible throughput loss when processing a stream of frames, since the IO time can be hidden by the relatively longer compute time. For instance, the throughput loss is only 3% for VGG as shown in Figure 4.8d, where buffer/tile size is 128 KB and SRAM size is 8 MB.

For those NNs seeing higher delay increase, the throughput loss is relatively higher, since the longer IO time cannot hidden by the relatively shorter compute time. Although MCU can reduce throughput loss by exploiting parallelism, but not much due to the limited number of buffers. For instance, the throughput loss for AlexNet and MobileNet is 15.7% and 46.4% as shown in Figure 4.8h and Figure 4.8l where buffer size is 128 KB and SRAM size is 8 MB.



(a) VGG, SRAM 512KB (b) VGG, SRAM 1MB (c) VGG, SRAM 4MB (d) VGG, SRAM 8MB

(e) AlexNet, SRAM 512KB (f) AlexNet, SRAM 1MB (g) AlexNet, SRAM 4MB (h) AlexNet, SRAM 8MB

(i) Mobilenet, SRAM 512KB (j) Mobilenet, SRAM 1MB (k) Mobilenet, SRAM 4MB (l) Mobilenet, SRAM 8MB

Figure4.8. Swapping throughput of NNs under diffent SRAM sizes and buffer/tile sizes.

***Implication*:**

Cross-frame (pipeline) parallelism cannot improve throughput much due to the limited number of buffers, even if increasing SRAM size.

A common pattern in an NN is that one or more compute-bound layers followed by one or more IO-bound layers, i.e. a *pipeline* with interleaved compute-bound and IO-bound stages. For instance, the AlexNet in Figure 4.2a, conv1-5 (compute-bound stage) is followed by fc6-8 (IO-bound stage). When executing NN on a sequence of frames, MCU can overlap IO/compute-bound stages of adjacent frames, hence hiding the IO delays that cannot be hidden at the layer/tile levels with each frame. As shown in Figure 4.9, MCU can swap for frame 0's FC layers while computing Frame 1's Conv layers, leading high MCU/IO utilization and throughput.

However, such parallelism that overlaps IO/compute-bound stages in adjacent frames cannot be fully exploited on MCUs with tiny SRAM due to the limited number of memory buffers. As Figure 4.8 shown, the throughput of VGG, AlexNet, and Mobilenet does not increase much as the SRAM size becomes larger. Because of the same reason as in latency above, the gap between the number of tasks and the number of buffers is too large (1000×). For instance, the number of IO tasks in MobileNet is 558770 (10 frames, 55877 IO tasks in each frame shown in Figure 4.7c) when buffer size is 16 KB, but the number of buffers only increase from 32 to 512 (1000× smaller than 558770) when SRAM size increases from 512 KB to 8 MB. The small number of buffers have been consumed by one frame, so other frames cannot get buffers to be executed in parallel.

***Implication*:**

Increasing buffer/tile size leads to higher throughput than increasing parallelism.

Same as the tradeoff in latency, given SRAM size: if the buffer/tile size is large (the number of buffer is small), the overall IO time is shot but parallelism is low; if the buffer/tile size is small (the number of buffer is large), the overall IO time is long but the parallelism is high. Overall, large buffer/tile size leads to higher throughput than small buffer/tile with high parallelism, especially for NNs that have more IO-bound layers. For instance, MobileNet

has more IO-bound layers, and its throughput increase 20× when buffer size increases from 16 KB to 512 KB (although parallelism drops due to less buffers) as shown in Figure 4.8l. While VGG and AlexNet have relatively less IO-bound layers, and their throughput does not change much when increasing buffer size, as shown in Figure 4.8d and Figure 4.8h. The reason is that parallelism is high when buffer/tile size is small, and overall IO time is short when buffer size is large.

4.5.4 Impact on Flash Durability

***Implication*:**

SD card sees negligible durability loss, and its lifetime could be years or tens of years with swapping.

The amount of data written to SD card per frame is not large because NN layers are read-most, and the write frequency is low due to the long execution time on slow MCU.

Modest write rate

For a given NN and SRAM size, the amount of data written to SD card is determined by the frame rate (reciprocal of delay per frame) and the amount of data to write per frame (upper bound is the sum of output feature maps of all layers), which have negative correlations: (1) for large NNs, frame rate is low but the amount of data to write per frame is large; (2) for small NNs, frame rate is high but the amount of data to write per frame is small. Therefore, no matter an NN is large or small, the data written per day won't be large. For instance, swapping writes only 2.0/2.8 GB for VGG16/AlexNet per day. Even for the extreme case, MobileNet, which has high frame rate and relatively large feature maps to write, swapping writes 123 GB per day.

SD card has long lifetime even with swapping

SD card is build up of many cells, which have limited write cycles [134]. As the capacity is becoming larger [135], the durability budget is keeping increasing. The study [136] keeps

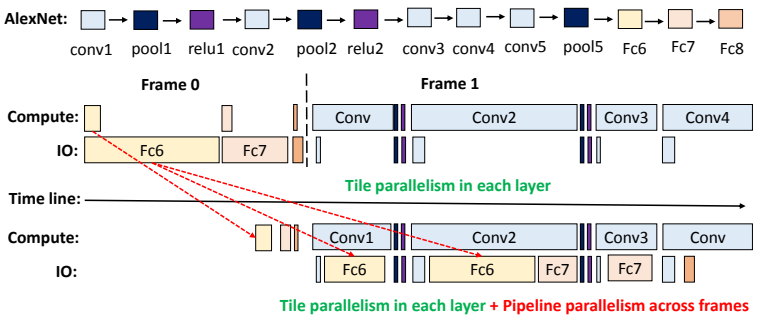


Figure4.9. AlexNet: tile parallelism for low delay and pipeline parallelism for high throughput.

writing 24/7 as fast as possible to 40 4 GB SD cards, and 1, 20, and 40 of 40 cards observe the first failures after writing 6.5 TB, 9 TB, and 12.5 TB of data to them. Based on their results, the first cell is only expected to fail on a 64 GB SD card after running MobileNet, AlexNet, and VGG16 for 2.4 – 4.5, 104 – 200, and 145 – 280 years, and 50% of cells fail (10K cycles per cell [137], [138]) only after running for 7.5, 328, and 460 years.

4.5.5 Impact on System Energy

***Implication*:**

Swapping adds modest energy consumption to an already busy MCU.

We estimate the *worst-case* energy overhead due to swapping. Our test platform is an STM32F746NG-Discovery board (ARM Cortex-M7 at 216 MHz; 340 KB SRAM) with an external power meter [139]. We run two benchmarks. (1) *in-core* emulates NN executions with an infinite amount of memory: it runs NN compute [133] for 1000 iterations. (2) *out-of-core* emulates NN executions with the most intensive IO traffic in parallel to the compute: it executes the same amount of compute with an IO thread repeatedly flushing data blocks to SD card Each data block is 100 KB (close to tile size); the flush is asynchronous using the MCU's DMA engine. Note that the IO traffic in real applications is less intensive (which will not keep writing all the time) than our benchmarks, so the energy we measure is the *worst-case* energy consumption that is higher than real cases.

Our measurement shows that: the additional IO workloads increases the system energy by 42%, from 0.07 Wh (in-core) to 0.10 Wh (out-of-core); the total execution time goes from 178 sec to 213 sec. Our obsevations are: (1) The *actual* energy overhead in out-of-core NNs is likely much less: while the *out-of-core* benchmark keeps IO always busy, the actual out-of-core NNs exercise IO intermittently (§4.2.1) because most NN layers are likely compute-bound. (2) We attribtute the modest energy overhead to the incremental nature of system energy: when an MCU-based device is already busy executing compute, its most power-hungry hardware – cores, interconnect, SRAM, and regulators – is already activated; executing IO, which activiates an SD card and the MMC controller in addition, adds to the energy but not much.

4.5.6 Out-of-core Data Security and Safety

Compared to storing NN data in on-chip SRAM, (temporarily) storing it off-chip is more vulnerable to physical attacks [140]: adversaries may learn or corrupt the data by tapping into the IO bus between MCU and the SD card, or the SD card itself. Fortunately, by encrypting NN data before swapping out, MCU can ensure the data to be confidential and integral; the overhead is linear to the data amount. Hardware crypto, such as for ASE [141], [142], is already common on modern MCUs. Its computation overhead is comparable to (or even less than) the least intensive NN compute (e.g. FC layers).

Compared to SRAM, SD cards are less durable. Yet, it is known that a SD card rarely fails as a whole but seeing a gradual increase number of corrupted cells over time [143]. Cell corruption is often silent, i.e. a read value simply differs from what was written last time. Fortunately, MCU can detects such failures with hash-based integrity checking. With specialized hardware on MCUs, computing hash is no more expensive than the least intensive NN compute [141]. Upon detection of bad cells, the MCU can recompute the most recent NN layer and recover the corrupted out-of-core data.

4.6 Related Work

Implications on model compression

Existing work on tinyML tries to run NNs on MCUs by reducing memory footprint, such as model compression [111]–[113], parameter quantization [114], designing tiny NNs from scratch [115], as well as automation of these procedures [116]. However, they give away model accuracy or generality at varying degrees. In order for an NN to fit into the MCU memory, the NN either becomes substantially inaccurate or too specialized. In contrast, our swapping solution doesn't incur accuracy and generality loss. Our solution boosts design freedom in tinyML, where memory limit was considered as the primary motivation for model compression. With the removal of such a limit, developers now have the choice of run large NNs without compression, retaining full model accuracy. Even in case of model compression is warranted, e.g. for faster NN execution, developers now have a wider selection of *baseline* NNs, including the ones with orders of higher memory footprints than MCUs.

Relation to prior swapping systems

Prior work enables out-of-core NN training with large batches on GPU/CPU memory systems [121], [144]–[147], but they cannot address the unique challenge on MCU that even a single layer exceeds main memory during NN inference. Prior work, e.g., Scratch-Pad [148] , proposes generic technique to swap data between SRAM and DRAM (not SD) for embedded devices. However, they don't leverage NN characteristics to optimize swapping, and they don't answer how swapping affects SD card lifetime, execution slowdown, energy consumption, and data security for NN applications. This chapter presents the first study on these questions and shows that swapping is feasible without much overhead.

Complement to existing inference framework

Tensorflow Lite Micro [149] is a framework for running NN inference on embedded devices. CMSIS-NN [133] provides optimized NN kernels for ARM Cortex-M MCUs. SONIC [150] supports intermittent computing for NN inference on MUCs. TVM [151] can generate op-

timized code for NNs on MCUs. However, none of them supports NNs whose memory footprints are larger than physical memory on MCUs. Our out-of-core solution is a complement to existing frameworks. It can be used in conjunction with them and expand their design space.

4.7 Summary

This chapter advocates enabling large NNs on tiny MCUs without losing accuracy by swapping data to SD card. With the parallel scheduler that overlaps IO and compute tasks to hide IO overhead, our study shows that none of SD card durability loss, execution slowdown, energy consumption, or data security is an issue. We find that an MCU with hundreds of KBs SRAM can execute NNs with a few hundreds MBs of memory footprint (a 1000× gap). Out-of-core execution expands the scope of NN applications on MCUs.

5. CONCLUSION

In this chapter, we first summarize the contributions of this thesis. Based on the lessons we learn from building StreamBox and StreamBox-HBM systems, we then provide general hints to system designs for future workloads on new hardware platforms.

5.1 Thesis Contributions

In this thesis, we demonstrate that: Novel runtime system designs not only can significantly speed up data analytics by exploiting emerging hardware platforms in the Cloud but also can enable data analytics on resource-constraint hardware platforms at the Edge.

5.1.1 System Support for Stream Processing on Cloud Hardware

We build StreamBox and StreamBox-HBM systems from scratch to speed up stream processing by exploiting multicore hardware and high-bandwidth hybrid memory hardware in the Cloud.

In StreamBox, our designs exploits the parallelism and memory hierarchy of modern multicore hardware. StreamBox executes a pipeline of transforms over records that may arrive out-of-order. The key contribution is to produce and manage abundant parallelism by generalizing out-of-order processing to out-of-order epoch processing, and by dynamically prioritize epochs to optimize latency. Experimental results show our system scales to a large number of cores and achieves throughput on-par with distributed engines on medium-size clusters.

In StreamBox-HBM, our designs exploit hybrid memory to achieve scalable high performance. We introduce a novel dynamic key/record pointer extraction into KPAs that minimizes the use of precious HBM capacity. We use sequential grouping algorithms on KPAs to balance limited capacity while exploiting high bandwidth. We design a runtime that manages parallelism and KPA placement in hybrid memories. Experimental results show that our system outperforms engines without KPA and with sequential-access algorithms by 7x and engines with random-access algorithms by an order of magnitude.

The designs of StreamBox/StreamBox-HBM provide a concrete example on how to optimize stream analytics on modern hardware for future researchers and engineers:

- Improving the performance of stream analytics requires systematic optimizations across software/hardware stack. This work achieves the best-reported performance by applying full-stack system optimizations, involving hardware features (multicore, hybrid memory, and SIMD), operating system, runtime system, network, memory management, scheduling, data structure, and algorithms;
- Improving the performance of stream analytics require identifying the root bottleneck of a wide range of operators, instead of individual operator. This work classifies all operators into Grouping and Reducing, and uses HBM to speed up all Grouping operators that dominate execution time, including Sort, Merge, Join, Select, Partitioning, etc.

No prior work on stream analytics has pushed performance to hardware limit or achieved such high performance (110 millions records/sec throughput under 1 second latency) on a single machine as StreamBox/StreamBox-HBM does. StreamBox/StreamBox can be easily integrated to existing distributed stream analytics frameworks as the execution backend on every single machine, such as Sparking Streaming and Apache Beam. They significantly improve the performance of stream analytics workloads.

5.1.2 System Support for Machine Learning Inference on Edge Hardware

We build SwapNN to enable large machine learning models on resource-constraint microcontrollers *without losing accuracy*, which was impossible before this work.

Different from prior algorithm-level solutions sacrificing accuracy as the cost, we investigate 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. We present a study showing that none of execution slowdown, storage wear out, energy consumption, or data security is a showstopper; the key benefit – MCUs being able to run large NNs with

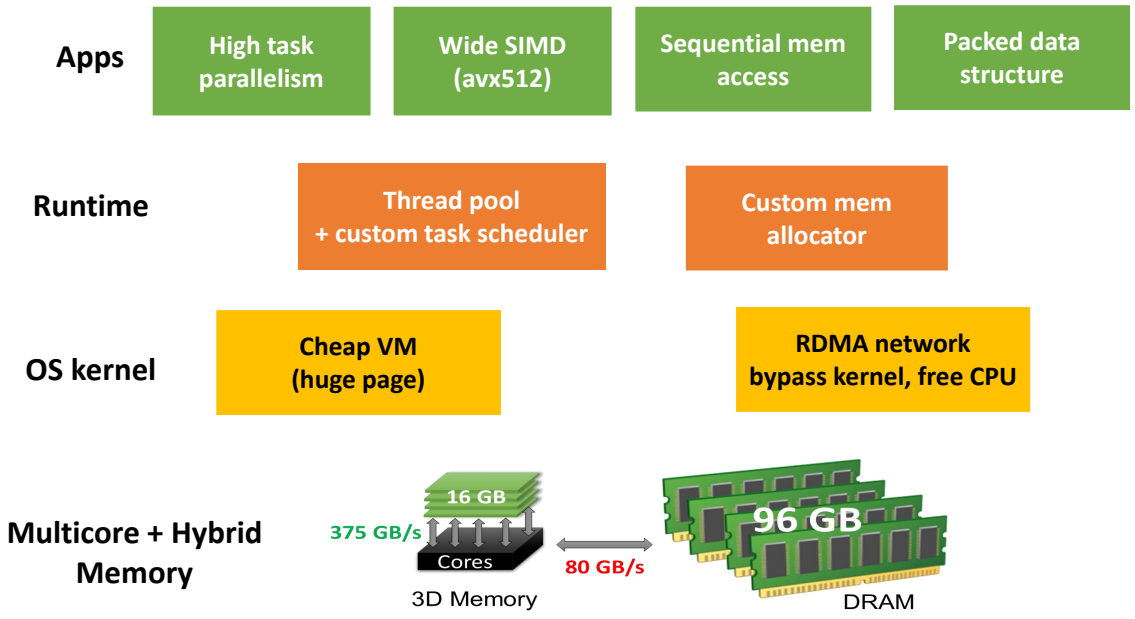


Figure5.1. Lessons on exploiting multicore and hybrid memory systems

full accuracy and generality – triumphs the overheads. Our findings suggest that MCUs can play a much greater role in edge intelligence.

SwapNN boosts design freedom in tinyML, where memory limit was considered as the primary motivation for model compression. With the removal of such a limit, developers now have the choice of run large NNs without compression, retaining full model accuracy. Even in case of model compression is warranted, e.g. for faster NN execution, developers now have a wider selection of *baseline* NNs, including the ones with orders of higher memory footprints than MCUs.

None of existing NN inference framework for MCUs can run NNs that have larger memory footprint than MCU's SRAM size as SwapNN does. SwapNN's out-of-core solution is a complement to existing frameworks. It can be used in conjunction with them and expand their design space. It expands the cope of NNs on MCUs.

5.2 General Lessons and Hints for Runtime System Designs

As illustrated in the thesis, building runtime systems for high-level applications by exploiting low-level hardware features is powerful to improve performance, reduce resource waste, and enable new use cases that were previously impossible. However, such runtime system designs require full-stack optimizations across hardware, OS, runtime, and applications. To help future system designs for new applications and new hardware, we summarize some general hints based on our experience in exploiting multicore and hybrid memory hardware. The key design choices in whole software/hardware stack are shown in Figure 5.1, and we will introduce them from the top app-level to the bottom hardware-level in the following subsections.

5.2.1 Apps: Algorithms Adapting to Hardware Changes

Our hint is that the best algorithms to choose may change when new hardware emerges, so system designers should rethink the algorithms on different hardware platforms. For example, both Sort and Hash can implement data Grouping. Prior work concludes Hash is best on DRAM, but we find Sort is best on HBM when we implement StreamBox-HBM. Because HBM employs a total wider bus with a wider SIMD vector, it changes the tradeoff for software.

For instance, based on this hint, we made a few choices in the application level to fully exploit the underlying multicore and hybrid memory hardware when building StreamBox and StreamBox-HBM: (1) high task parallelism to fully utilize under multicore architecture; (2) wide SIMD to make sure that CPU cores can generate enough outstanding memory requests to utilize the high bandwidth of HBM; (3) sequential memory access to ensure cache locality and row buffer locality in HBM and DRAM; (4) packed data structure to reduce the memory consumption for the small capacity of HBM.

5.2.2 Runtime: Better Managing Resources than General Hardware and OS

Our hint is that runtime systems better manage resources than general hardware and OS, because they can leverage the knowledge of both applications' unique demands and hardware's unique features. For example, both hardware and OS can manage hybrid memory transparently on Intel Knights Landing, but they waste bandwidth either because of moving all data from DRAM to HBM for every access or not being aware of bandwidth/capacity differences or applications' memory access pattern. In contrast, StreamBox-HBM runtime fully controls data placement based on the knowledge of stream analytics and hybrid memory, and it achieves significant speedup comparing to hardware or OS approaches.

For instance, based on this hint, we make the following choices in StreamBox and StreamBox-HBM: (1) customized memory allocator, instead of OS's default, to dynamically allocate memory from HBM and DRAM based on HBM's capacity constraint and DRAM's bandwidth constraint, and to decide the granularity of memory allocation based on application's memory access. (2) customized thread pool and task scheduler, instead of OS's default scheduler, to map tasks to CPU cores based on application's knowledge and demands, e.g., the emergency of tasks.

5.2.3 OS: Configuring Kernel Parameters Accordingly

Our hint is that the kernel parameters should be configured appropriately based on application's demands and hardware features. For example, Linux supports general 4KB page and huge page (e.g., 2MB). For data intensive workloads that continuously allocate and free large memory chunks, huge page is a better choice to reduce the overhead of memory allocation and address translation.

For instance, based on this hint, we made the following choices in StreamBox-HBM: (1) configuring OS kernel to enable huge page and mapping all physical memory into user space without freeing them, so that runtime system can fully control memory management to avoid the overhead of memory mapping/unmapping or allocation/deallocation. (2) bypassing OS kernel using RDMA to free CPU from transferring IO data.

5.2.4 Hardware: Choosing Hardware Based on Applications' Demand

Our hint is that choosing underlying hardware should consider the unique demand of high-level applications. For instance, for data intensive applications that have insatiable demands for memory and performance, high-bandwidth memory and multicore CPUs are intuitively good matches. Although they cannot not improve applications performance out of the box, but advanced runtime system designs can solve this problem as we demonstrated in the thesis.

REFERENCES

[1] J. Bulao, *How much data is created every day in 2021?* 2021. [Online]. Available: https://techjury.net/blog/how-much-data-is-created-every-day/#gref.

[2] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley and F. X. Lin, 'Streambox: Modern stream processing on a multicore machine,' in *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'17, USENIX Association, 2017.

[3] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker and I. Stoica, 'Discretized streams: Fault-tolerant streaming computation at scale,' in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ACM, 2013, pp. 423–438.

[4] Apache, *Beam*, https://beam.apache.org/, 2017.

[5] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley and F. X. Lin, 'Streambox-hbm: Stream analytics on high bandwidth hybrid memory,' in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2019, pp. 167–181.

[6] H. Miao and F. X. Lin, 'Enabling large neural networks on tiny microcontrollers with swapping,' *arXiv preprint arXiv:2101.08744*, 2021.

[7] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, 'The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing,' *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[8] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou and L. Zhou, 'Streamscope: Continuous reliable distributed processing of big data streams,' in *Proc. of NSDI*, 2016, pp. 439–454.

[9] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu and Z. Zhang, 'Timestream: Reliable stream computation in the cloud,' in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, Prague, Czech Republic: ACM, 2013, pp. 1–14, ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465353. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465353.

[10] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson and D. Maier, 'Out-of-order processing: A new architecture for high-performance stream systems,' *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 274–288, 2008.

[11] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom and S. Whittle, 'Millwheel: Fault-tolerant stream processing at internet scale,' *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1033–1044, Aug. 2013, ISSN: 2150-8097. DOI: 10.14778/2536222.2536229. [Online]. Available: http://dx.doi.org/10.14778/2536222.2536229.

[12] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger and J. Wernsing, 'Trill: A high-performance incremental query processor for diverse analytics,' *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 401–412, 2014.

[13] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham and M. Abadi, 'Naiad: A timely dataflow system,' in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 439–455, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522738. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522738.

[14] C. Desrochers, *Moodycamel::concurrentqueue*, 2016. [Online]. Available: https://github.com/cameron314/concurrentqueue.

[15] D. Alistarh, J. Kopinsky, J. Li and N. Shavit, 'The spraylist: A scalable relaxed priority queue,' *SIGPLAN Not.*, vol. 50, no. 8, pp. 11–20, Jan. 2015, ISSN: 0362-1340. DOI: 10.1145/2858788.2688523. [Online]. Available: http://doi.acm.org/10.1145/2858788.2688523.

[16] A. Gurtovoyi and D. Abrahamsi, *Boost c++ libraries*, http://www.boost.org/, 2017.

[17] Intel, *Intel threading building blocks*, 2017. [Online]. Available: https://software.intel.com/en-us/intel-tbb.

[18] Facebook, *Folly*, 2017. [Online]. Available: https://github.com/facebook/folly#folly-facebook-open-source-library.

[19] J. E. David Goldblatt Dave Watson, *Jemalloc memory allocator*, http://http://jemalloc.net/, 2017.

[20] Intel, *Scalable memory allocator*, https://www.threadingbuildingblocks.org/tutorial-intel-tbb-scalable-memory-allocator, 2017.

[21] M. Goyal, B. Fan, X. Li, D. G. Andersen and M. Kaminsky, *Libcuckoo*, 2017. [Online]. Available: https://github.com/efficient/libcuckoo.

[22] V. Leis, P. Boncz, A. Kemper and T. Neumann, 'Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age,' in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, ACM, 2014, pp. 743–754.

[23] C. Balkesen, J. Teubner, G. Alonso and M. T. Özsu, 'Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware,' in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, IEEE, 2013, pp. 362–373.

[24] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang *et al.*, 'Bigdatabench: A big data benchmark suite from internet services,' in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, IEEE, 2014, pp. 488–499.

[25] M. Hart, *Free ebooks by project gutenberg*, , 2017.

[26] P. S. Eugene Kharitonov, *Yandex: Personalized web search challenge*, https://www.kaggle.com/c/yandex-personalized-web-search-challenge/data, 2017.

[27] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen *et al.*, 'Pingmesh: A large-scale system for data center network latency measurement and analysis,' *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 139–152, 2015.

[28] Z. Cheng, J. Caverlee and K. Lee, 'You are where you tweet: A content-based approach to geo-locating twitter users,' in *Proceedings of the 19th ACM international conference on Information and knowledge management*, ACM, 2010, pp. 759–768.

[29] C. Cranor, T. Johnson, O. Spataschek and V. Shkapenyuk, 'Gigascope: A stream database for network applications,' in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ACM, 2003, pp. 647–651.

[30] P. A. Tucker, D. Maier, T. Sheard and L. Fegaras, 'Exploiting punctuation semantics in continuous data streams,' *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 3, pp. 555–568, 2003.

[31] Oracle®, *Stream explorer*, 2017. [Online]. Available: http://bit.ly/1L6tKz3.

[32] M. C. Stanley Zdonik Michael Stonebraker, *Streambase systems*, 2017. [Online]. Available: http://www.tibco.com/products/tibco-streambase.

[33] EsperTech, *Esper*, 2017. [Online]. Available: http://www.espertech.com/esper/.

[34] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa and P. Pietzuch, 'Saber: Window-based hybrid stream processing for heterogeneous architectures,' in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, San Francisco, California, USA: ACM, 2016, pp. 555–569, ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882906. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882906.

[35] EsperTech, *Esper faq*, 2017. [Online]. Available:
.

[36] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin *et al.*, 'Aurora: A data stream management system,' in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ACM, 2003, pp. 666–666.

[37] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss and M. A. Shah, 'Telegraphcq: Continuous dataflow processing,' in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ACM, 2003, pp. 668–668.

[38] D. Maier, J. Li, P. Tucker, K. Tufte and V. Papadimos, 'Semantics of data streams and operators,' in *International Conference on Database Theory*, Springer, 2005, pp. 37–52.

[39] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger and J. Wernsing, 'Trill: A high-performance incremental query processor for diverse analytics,' *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 401–412, 2014.

[40] Twitter, *Heron*, https://twitter.github.io/heron/, 2017.

[41] A. Roy, I. Mihailovic and W. Zwaenepoel, 'X-stream: Edge-centric graph processing using streaming partitions,' in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 472–488, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522740. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522740.

[42] A. Kyrola, G. Blelloch and C. Guestrin, 'Graphchi: Large-scale graph computation on just a PC,' in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, Hollywood, CA, USA: USENIX Association, 2012, pp. 31–46, ISBN: 978-1-931971-96-6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387884.

[43] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht and I. Stoica, 'Drizzle: Fast and adaptable stream processing at scale,' in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, Shanghai, China: ACM, 2017, pp. 374–389, ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132750. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132750.

[44] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, 'The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing,' *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[45] H. Solutions, *Tpc-h*, http://www.tpc.org/tpch/, Last accessed: July 25, 2018.

[46] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li and B. Qiu, 'Bigdatabench: A big data benchmark suite from internet services,' in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb. 2014, pp. 488–499. DOI: 10.1109/HPCA.2014.6835958.

[47] AMPLab, *Amplab big data benchmark*, Last accessed: July 25, 2018. [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/#.

[48] R. Xie, *Malware detection*, https://www.endgame.com/blog/technical-blog/data-science-security-using-passive-dns-query-data-analyze-malware, Last accessed: Jan 25, 2019.

[49] Intel, *Knights Landing, the Next Generation of Intel Xeon Phi*, http://www.enterprisetech.com/2014/11/17/enterprises-get-xeon-phi-roadmap/, Last accessed: Dec. 08, 2014.

[50] nVIDIA, *Nvidia titan v*, https://www.nvidia.com/en-us/titan/titan-v/, 2018.

[51] Xilinx, *Xilinx virtex ultrascale+*, https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html, 2018.

[52] Google, *Google clout tpu*, https://cloud.google.com/tpu/, 2018.

[53] A. Li, W. Liu, M. R. B. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez and S. L. Song, 'Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels,' in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, Denver, Colorado: ACM, 2017, 26:1–26:14, ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126931. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126931.

[54] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure and S. Markidis, 'Exploring the performance benefit of hybrid memory system on hpc environments,' in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 683–692. DOI: 10.1109/IPDPSW.2017.115.

[55] Y. You, A. Buluç and J. Demmel, 'Scaling deep learning on gpu and knights landing clusters,' in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, Denver, Colorado: ACM, 2017, 9:1–9:12, ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126912. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126912.

[56] *Apache Beam*. [Online]. Available: https://beam.apache.org/.

[57] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl and K. Tzoumas, 'Apache flink: Stream and batch processing in a single engine,' *Data Engineering*, p. 28, 2015.

[58] J. Jeffers, J. Reinders and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[59] Yahoo! *Benchmarking Streaming Computation Engines at Yahoo!* https://yahooeng.tumblr.com/post/135321837876/, Last accessed: May. 01, 2018.

[60] C. Balkesen, G. Alonso, J. Teubner and M. T. Özsu, 'Multi-core, main-memory joins: Sort vs. hash revisited,' *Proc. VLDB Endow.*, vol. 7, no. 1, pp. 85–96, Sep. 2013, ISSN: 2150-8097. DOI: 10.14778/2732219.2732227. [Online]. Available: http://dx.doi.org/10.14778/2732219.2732227.

[61] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas and P. Dubey, 'Sort vs. hash revisited: Fast join implementation on modern multi-core cpus,' *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009, ISSN: 2150-8097. DOI: 10.14778/1687553.1687564. [Online]. Available: https://doi.org/10.14778/1687553.1687564.

[62] O. Polychroniou, A. Raghavan and K. A. Ross, 'Rethinking simd vectorization for in-memory databases,' in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: ACM, 2015, pp. 1493–1508, ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2747645. [Online]. Available: http://doi.acm.org/10.1145/2723372.2747645.

[63] P. A. Boncz, M. Zukowski and N. Nes, 'Monetdb/x100: Hyper-pipelining query execution.,' in *Cidr*, vol. 5, 2005, pp. 225–237.

[64] P.-A. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu and M. Saubhasik, 'Enhancements to sql server column stores,' in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, New York, USA: ACM, 2013, pp. 1159–1168, ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2463708. [Online]. Available: http://doi.acm.org/10.1145/2463676.2463708.

[65] T. J. Lehman and M. J. Carey, *Query processing in main memory database management systems*, 2. ACM, 1986, vol. 15.

[66] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray and D. Lomet, 'Alphasort: A risc machine sort,' *SIGMOD Rec.*, vol. 23, no. 2, pp. 233–242, May 1994, ISSN: 0163-5808. DOI: 10.1145/191843.191884. [Online]. Available: http://doi.acm.org/10.1145/191843.191884.

[67] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm and L. Zhang, 'Db2 with blu acceleration: So much more than just a column store,' *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1080–1091, Aug. 2013, ISSN: 2150-8097. DOI: 10.14778/2536222.2536233. [Online]. Available: http://dx.doi.org/10.14778/2536222.2536233.

[68] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran and S. Zdonik, 'C-store: A column-oriented dbms,' in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05, Trondheim, Norway: VLDB Endowment, 2005, pp. 553–564, ISBN: 1-59593-154-6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083592.1083658.

[69] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot and D. Pnevmatikatos, 'The mondrian data engine,' in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, Toronto, ON, Canada: ACM, 2017, pp. 639–651, ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080233. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080233.

[70] JEDEC, *High bandwidth memory (hbm) dram. standard no. jesd235*, 2013.

[71] JEDEC, *High bandwidth memory 2. standard no. jesd235a*, 2016.

[72] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung and S. Hong, '25.2 a 1.2v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv,' in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb. 2014, pp. 432–433. DOI: 10.1109/ISSCC.2014.6757501.

[73] E. J. Fluhr, J. Friedrich, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, A. Hall, D. Hogenmiller, F. Malgioglio, R. Nett, J. Paredes, J. Pille, D. Plass, R. Puri, P. Restle, D. Shan, K. Stawiasz, Z. T. Deniz, D. Wendel and M. Ziegler, 'Power8: A 12-core server-class processor in 22nm soi with 7.6tb/s off-chip bandwidth,' in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb. 2014, pp. 96–97. DOI: 10.1109/ISSCC.2014.6757353.

[74] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza and T. Burton, 'Haswell: The fourth-generation intel core processor,' *IEEE Micro*, no. 2, pp. 6–20, 2014.

[75] X. Cheng, B. He, X. Du and C. T. Lau, 'A study of main-memory hash joins on many-core processor: A case with intel knights landing architecture,' in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17, Singapore, Singapore: ACM, 2017, pp. 657–666, ISBN: 978-1-4503-4918-5. DOI: 10.1145/3132847.3132916. [Online]. Available: http://doi.acm.org/10.1145/3132847.3132916.

[76] C. Wang, T. Coa, J. Zigman, F. Lv, Y. Zhang and X. Feng, 'Efficient management for hybrid memory in managed language runtime,' in *IFIP International Conference on Network and Parallely Computing (NPC)*, 2016.

[77] W. Wei, D. Jiang, S. A. McKee, J. Xiong and M. Chen, 'Exploiting program semantics to place data in hybrid memory,' in *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, 2015, ISBN: 978-1-4673-9524-3.

[78] W. Zhang and T. Li, 'Exploring phase change memory and 3d die-stacking for power-/thermal friendly, fast and durable memory architectures,' in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009, ISBN: 978-0-7695-3771-9.

[79] A. Arasu, S. Babu and J. Widom, 'The cql continuous query language: Semantic foundations and query execution,' *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006, ISSN: 1066-8888. DOI: 10.1007/s00778-004-0147-z. [Online]. Available: http://dx.doi.org/10.1007/s00778-004-0147-z.

[80] *Intel Performance Counter Monitor - A better way to measure CPU utilization*, Last accessed: May. 01, 2017. [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[81] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley and F. X. Lin, *Streambox code*, https://engineering.purdue.edu/~xzl/xsel/p/streambox/index.html, Last accessed: July 25, 2018.

[82] G. Pekhimenko, C. Guo, M. Jeon, R. Huang and L. Zhou, 'Tersecades: Efficient data compression in stream processing,' in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, USENIX Association, 2018.

[83] Data Artisians, *The Curious Case of the Broken Benchmark: Revisiting Apache Flink vs. Databricks Runtime*, https://data-artisans.com/blog/curious-case-broken-benchmark-revisiting-apache-flink-vs-databricks-runtime, Last accessed: May. 01, 2018.

[84] DataBricks, *Benchmarking Structured Streaming on Databricks Runtime Against State-of-the-Art Streaming Systems*, https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-art-streaming-systems.html, Last accessed: May. 01, 2018.

[85] Z. Jerzak and H. Ziekow, 'The debs 2014 grand challenge,' in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14, Mumbai, India: ACM, 2014, pp. 266–269, ISBN: 978-1-4503-2737-4. DOI: 10.1145/2611286.2611333. [Online]. Available: http://doi.acm.org/10.1145/2611286.2611333.

[86] M.-C. Albutiu, A. Kemper and T. Neumann, 'Massively parallel sort-merge joins in main memory multi-core database systems,' *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1064–1075, Jun. 2012, ISSN: 2150-8097. DOI: 10.14778/2336664.2336678. [Online]. Available: http://dx.doi.org/10.14778/2336664.2336678.

[87] iMatix Corporation, *Zeromq*, http://zeromq.org/, 2018.

[88] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, 'Storm@ twitter,' in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, ACM, 2014, pp. 147–156.

[89] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa and P. Pietzuch, 'Saber: Window-based hybrid stream processing for heterogeneous architectures,' in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, San Francisco, California, USA: ACM, 2016, pp. 555–569, ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882906. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882906.

[90] M. Yip and T. Company, *Rapidjson*, https://github.com/Tencent/rapidjson, Last accessed: July 25, 2018.

[91] Google, *Google protocol buffers*, https://developers.google.com/protocol-buffers/, Last accessed: July 25, 2018.

[92] Jan, *String-to-uint64*, http://jsteemann.github.io/blog/2016/06/02/fastest-string-to-uint64-conversion-method/, Last accessed: Jan 25, 2019.

[93] A. Hagiescu, W.-F. Wong, D. F. Bacon and R. Rabbah, 'A computing origami: Folding streams in fpgas,' in *Proceedings of the 46th Annual Design Automation Conference*, ACM, 2009, pp. 282–287.

[94] S. Rajadurai, J. Bosboom, W.-F. Wong and S. Amarasinghe, 'Gloss: Seamless live reconfiguration and reoptimization of stream programs,' in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, Williamsburg, VA, USA: ACM, 2018, pp. 98–112, ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173170. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173170.

[95] B. Bramas, 'Fast sorting algorithms using avx-512 on intel knights landing,' *arXiv preprint arXiv:1704.08579*, 2017.

[96] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson and K. Schwan, 'Data tiering in heterogeneous memory systems,' in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, London, United Kingdom: ACM, 2016, 15:1–15:16, ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901344. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901344.

[97] N. Agarwal and T. F. Wenisch, 'Thermostat: Application-transparent page management for two-tiered main memory,' in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, Xi'an, China: ACM, 2017, pp. 631–644, ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037706. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037706.

[98] T. D. Doudali and A. Gavrilovska, 'Comerge: Toward efficient data placement in shared heterogeneous memory systems,' in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17, Alexandria, Virginia: ACM, 2017, pp. 251–261, ISBN: 978-1-4503-5335-9. DOI: 10.1145/3132402.3132418. [Online]. Available: http://doi.acm.org/10.1145/3132402.3132418.

[99] S. Wen, L. Cherkasova, F. X. Lin and X. Liu, 'Profdp: A lightweight profiler to guide data placement in heterogeneous memory systems,' in *Proceedings of the 32th ACM on International Conference on Supercomputing*, ser. ICS '18, Beijing, China: ACM, 2018.

[100] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, 'Wisckey: Separating keys from values in ssd-conscious storage,' in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, Santa Clara, CA: USENIX Association, 2016, pp. 133–148, ISBN: 978-1-931971-28-7. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu.

[101] F. Xia, D. Jiang, J. Xiong and N. Sun, 'Hikv: A hybrid index key-value store for dram-nvm memory systems,' in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, 2017, pp. 349–362, ISBN: 978-1-931971-38-6. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia.

[102] *An introduction to tinyml*, 2020. [Online]. Available: https://towardsdatascience.com/an-introduction-to-tinyml-4617f314aa79.

[103] *Nuru ai expansion: Supporting farmers to diagnose crop diseases*, 2020. [Online]. Available: https://blog.plantwise.org/2020/03/13/nuru-ai-expansion-supporting-farmers-to-diagnose-crop-diseases/.

[104] *Stmicroelectronics stm32 family*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/STM32.

[105] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, 'Going deeper with convolutions,' in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[106] K. Siu, D. M. Stuart, M. Mahmoud and A. Moshovos, 'Memory requirements for convolutional neural network hardware accelerators,' in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2018, pp. 111–121.

[107] K. Simonyan and A. Zisserman, 'Very deep convolutional networks for large-scale image recognition,' *arXiv preprint arXiv:1409.1556*, 2014.

[108] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, 'Going deeper with convolutions,' in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[109] *The role of srams in nextgen iot and wearable embedded designs*, 2014. [Online]. Available: https://www.embedded.com/the-role-of-srams-in-nextgen-iot-and-wearable-embedded-designs/.

[110] D. Blalock, J. J. G. Ortiz, J. Frankle and J. Guttag, 'What is the state of neural network pruning?' In *MLSys*, 2020.

[111] M. Zhu and S. Gupta, 'To prune, or not to prune: Exploring the efficacy of pruning for model compression,' *arXiv preprint arXiv:1710.01878*, 2017.

[112] S. Han, H. Mao and W. J. Dally, 'Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,' *arXiv preprint arXiv:1510.00149*, 2015.

[113] H. Li, A. Kadav, I. Durdanovic, H. Samet and H. P. Graf, 'Pruning filters for efficient convnets,' *arXiv preprint arXiv:1608.08710*, 2016.

[114] S. Gupta, A. Agrawal, K. Gopalakrishnan and P. Narayanan, 'Deep learning with limited numerical precision,' in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

[115] H. Yang, M. Fritzsche, C. Bartz and C. Meinel, 'Bmxnet: An open-source binary neural network implementation based on mxnet,' in *Proceedings of the 25th ACM international conference on Multimedia*, ACM, 2017, pp. 1209–1212.

[116] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan and S. Han, 'Mcunet: Tiny deep learning on iot devices,' *arXiv preprint arXiv:2007.10319*, 2020.

[117] M. Xu, X. Zhang, Y. Liu, G. Huang, X. Liu and F. X. Lin, 'Approximate query service on autonomous iot cameras,' in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 191–205.

[118] H. Shen, S. Han, M. Philipose and A. Krishnamurthy, 'Fast video classification via adaptive cascading of deep models,' in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 3646–3654.

[119] M. Xu, F. Qian, Q. Mei, K. Huang and X. Liu, 'Deeptype: On-device deep learning for input personalization service with minimal privacy concern,' *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 4, pp. 1–26, 2018.

[120] M. Alwani, H. Chen, M. Ferdman and P. Milder, 'Fused-layer cnn accelerators,' in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Press, 2016, p. 22.

[121] C.-C. Huang, G. Jin and J. Li, 'Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping,' in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1341–1355.

[122] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam, 'Mobilenets: Efficient convolutional neural networks for mobile vision applications,' *arXiv preprint arXiv:1704.04861*, 2017.

[123] A. Krizhevsky, I. Sutskever and G. E. Hinton, 'Imagenet classification with deep convolutional neural networks,' in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[124] *Floating point operations per second*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/FLOPS.

[125] *Arm cortex-m*, 2020. [Online]. Available:
.

[126] *Microsd card benchmarks*, 2020. [Online]. Available: https://www.pidramble.com/wiki/benchmarks/microsd-cards.

[127] *Roofline model*, 2020. [Online]. Available: .

[128] S. albanie, *Estimates of memory consumption and flop counts for various convolutional neural networks.* 2021. [Online]. Available: https://github.com/albanie/convnet-burden.

[129] K. He, X. Zhang, S. Ren and J. Sun, 'Deep residual learning for image recognition,' in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[130] *Stm32 32-bit arm cortex mcu*, 2020. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html.

[131] *Nxp general purpose microcontrollers*, 2020. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus:GENERAL-PURPOSE-MCUS.

[132] Y. Zhang, N. Suda, L. Lai and V. Chandra, 'Hello edge: Keyword spotting on microcontrollers,' *arXiv preprint arXiv:1711.07128*, 2017.

[133] L. Lai, N. Suda and V. Chandra, 'Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus,' *arXiv preprint arXiv:1801.06601*, 2018.

[134] *Kingston flash memory guide*, 2015. [Online]. Available: .

[135] *History and evolution of memory cards*, 2020. [Online]. Available: https://koofr.eu/blog/posts/history-and-evolution-of-memory-cards.

[136] *Sd cart testing*, 2020. [Online]. Available: https://support.embeddedarm.com/support/solutions/articles/22000202866-sd-card-testing.

[137] *Every thing you need to know about slc, mlc, and tlc nand flash*, 2015. [Online]. Available: https://www.mydigitaldiscount.com/everything-you-need-to-know-about-slc-mlc-and-tlc-nand-flash.html.

[138] *Transcend industrial temp microsd 64 gb*, 2020. [Online]. Available: .

[139] *Usb c power meter tester*, 2020. [Online]. Available: .

[140] *The exploration and exploitation of an sd memory card*, 2020. [Online]. Available: http://bunniefoo.com/bunnie/sdcard-30c3-pub.pdf.

[141] *Performance of state-of-the-art cryptography on arm-based microprocessors*, 2020. [Online]. Available: https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/presentations/session7-vincent.pdf.

[142] P. Schwabe and K. Stoffelen, 'All the aes you need on cortex-m3 and m4,' in *International Conference on Selected Areas in Cryptography*, Springer, 2016, pp. 180–194.

[143] *Reliable sd-based block storage*, 2017. [Online]. Available: https://support.embeddedarm.com/support/solutions/articles/22000202867-reliable-sd-based-block-storage.

[144] A. Hayakawa and T. Narihira, 'Out-of-core training for extremely large-scale neural networks with adaptive window-based scheduling,' *arXiv preprint arXiv:2010.14109*, 2020.

[145] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar and S. W. Keckler, 'Vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design,' in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–13.

[146] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins *et al.*, 'Dynamic control flow in large-scale machine learning,' in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–15.

[147] T. Jin and S. Hong, 'Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization,' in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 835–847.

[148] A. Dominguez, S. Udayakumaran and R. Barua, 'Heap data allocation to scratch-pad memory in embedded systems,' *Journal of Embedded Computing*, vol. 1, no. 4, pp. 521–540, 2005.

[149] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev *et al.*, 'Tensorflow lite micro: Embedded machine learning on tinyml systems,' *arXiv preprint arXiv:2010.08678*, 2020.

[150] G. Gobieski, B. Lucia and N. Beckmann, 'Intelligence beyond the edge: Inference on intermittent embedded systems,' in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2019, pp. 199–213.

[151] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, 'Tvm: An automated end-to-end optimizing compiler for deep learning,' in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.