StreamBox-HBM: Stream Analytics on High Bandwidth Hybrid Memory

Hongyu Miao
Purdue ECE

Myeongjae Jeon
UNIST

Gennady Pekhimenko
University of Toronto

Kathryn S. McKinley
Google

Felix Xiaozhu Lin
Purdue ECE

Abstract

Stream analytics has an insatiable demand for memory and performance. Emerging *hybrid* memories combine commodity DDR4 DRAM with 3D-stacked High Bandwidth Memory (HBM) DRAM to meet such demands. However, achieving this promise is challenging because (1) HBM is capacity-limited and (2) HBM boosts performance best for sequential access and high parallelism workloads. At first glance, stream analytics appears a particularly poor match for HBM because they have high capacity demands and data grouping operations, their most demanding computations, use random access.

This paper presents the design and implementation of StreamBox-HBM, a stream analytics engine that exploits hybrid memories to achieve scalable high performance. StreamBox-HBM performs data grouping with sequential access sorting algorithms in HBM, in contrast to random access hashing algorithms commonly used in DRAM. StreamBox-HBM solely uses HBM to store *Key Pointer Array* (KPA) data structures that contain only partial records (keys and pointers to full records) for grouping operations. It dynamically creates and manages prodigious data and pipeline parallelism, choosing when to allocate KPAs in HBM. It dynamically optimizes for both the high bandwidth and limited capacity of HBM, and the limited bandwidth and high capacity of standard DRAM.

StreamBox-HBM achieves 110 million records per second and 238 GB/s memory bandwidth while effectively utilizing all 64 cores of Intel's Knights Landing, a commercial server with hybrid memory. It outperforms stream engines with sequential access algorithms without KPAs by 7× and stream engines with random access algorithms by an order of magnitude in throughput. To the best of our knowledge, StreamBox-HBM is the first stream engine optimized for hybrid memories.

CCS Concepts • **Computer systems organization → Multicore architectures**; **Heterogeneous (hybrid) systems**; • **Information systems → DBMS engine architectures**.

Keywords KPA; data analytics; stream processing; high bandwidth memory; hybrid memory; multicore

ACM Reference Format:
Hongyu Miao, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2019. StreamBox-HBM: Stream Analytics on High Bandwidth Hybrid Memory. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304031

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) [13, 44, 60, 71]. 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 [4] and Microsoft Trill [13]. Grouping operations dominate queries in TPC-H (18 of 22) [56], BigDataBench (10 of 19) [62], AMPLab Big Data Benchmark (3 of 4) [7], and even Malware Detection [66]. 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 [29]), GPUs (NVIDIA Titan V [46]), FPGAs (Xilinx Virtex UltraScale+ [67]), and Cloud TPUs (v2 and v3 [24]) 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 [29]. Although researchers have achieved substantial improvements for high performance computing [40, 50] and machine learning [70] 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 (§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 [1, 5, 12, 44, 71].

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 [33] 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 [12] on the popular YSB benchmark [68] where StreamBox-HBM achieves 18× higher throughput per core. Much prior work reports results without data ingress [13, 44]. As far as we know, StreamBox-HBM achieves the best reported records per second for streaming *with ingress* on a single machine.

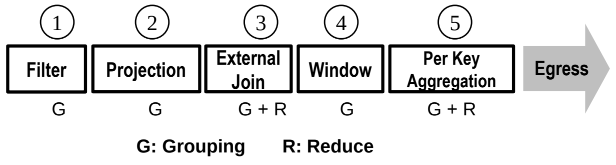
The key contributions are as follows. (1) New empirical results find on real hardware that sequential sorting algorithms for grouping are best for HBM, in contrast to DRAM, where random hashing algorithms are best [9, 35, 51]. Based on this finding, we optimize grouping computations with sequential algorithms. (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 [10, 13, 37, 39, 47, 54, 58], 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 [18]. 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.

2 Background & Motivation

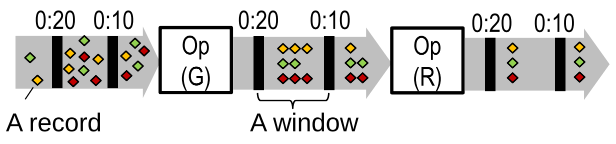
This section presents background on our stream analytics programming model, runtime, and High Bandwidth Memory (HBM). Motivating results explore `GroupBy` implementations with sorting and hashing on HBM. We find merge-sort exploits HBM's high memory bandwidth with sequential access patterns and high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

2.1 Modern Stream Analytics

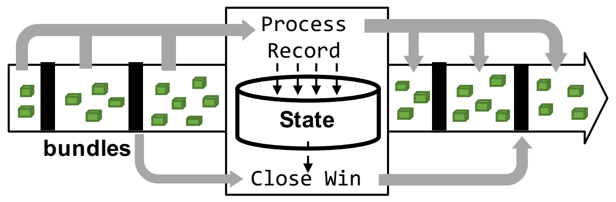
Programming model We adopt the popular Apache Beam programming model [1] used by stream engines such



(a) Pipeline of Yahoo streaming benchmark (YSB) [68] 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) & (5). The pipeline will serve as our running example for design and evaluation (§4 and §7).



(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

Figure 1. Example streaming data and computations

as Flink [12], Spark Streaming [71], and Google Cloud Dataflow [5]. These engines all use declarative stream operators that group, reduce, or do both on stream data such as those in Table 1. 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);
```

Listing 1. Example Stream Program. It sums up values for each key in every 1-second fixed-size window.

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

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

Streaming computations: grouping & reduction The declarative operators in Table 1 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 1a. In most pipelines, grouping dominates total execution time.

Stream execution model Figure 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 [41]. 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 [44]. Figure 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.

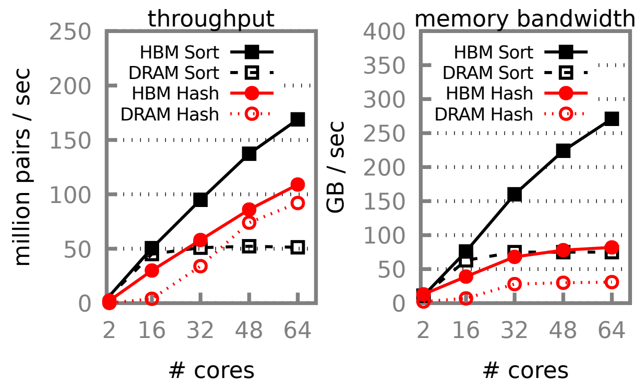


Figure 2. GroupBy on HBM and DRAM operating on 100M key/value records with about 100 values per key. Keys and values are 64-bit random integers. Sort leverages HBM bandwidth with sequential access and outperforms Hash on HBM.

2.2 Exploiting HBM

Modern HBM stacks up to 8 DRAM dies in special purpose silicon chips [31, 32]. Compared to normal DRAM, HBM offers (1) 5–10× higher bandwidth, (2) 5–10× smaller capacity due to cost and power [31, 32, 38], 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 [22, 26, 29]. 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). (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 (§ 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 [35], and implement Sort from a fast implementation [14] and hand-optimize it with AVX-512. Our *Hash* is 4× faster (not shown) than a popular, fast hash table *not* optimized for KNL [21]. Both implementations achieve state-of-the-art performance on KNL.

Figure 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 *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 [38]. (4) HBM favors sequential-access algorithms even though they incur higher algorithmic complexity.

Prior work explored tradeoffs for *Sort* and *Hash* on DRAM [9, 35, 51], 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 [1, 12, 13, 44, 71]. (2) They lack mechanisms for managing data and intermediate results between HBM and DRAM. Although the hardware or OS could manage data placement [61, 63, 72], 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 [44], most engines generate pipeline parallelism, but do not generate sufficient total parallelism to saturate HBM bandwidth.

3 Overview of StreamBox-HBM

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

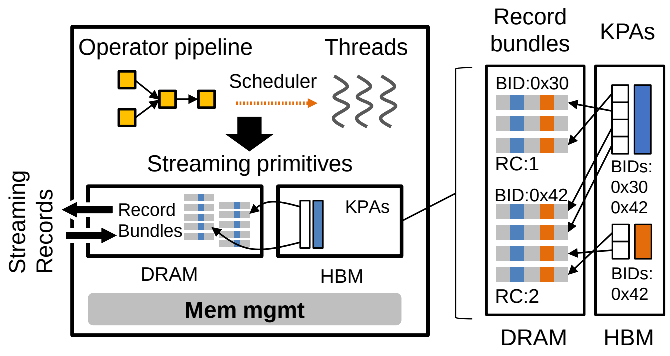


Figure 3. An overview of StreamBox-HBM using record bundles and KPAs. RC: reference count; BID: bundle ID.

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.

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 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 [12, 13, 44, 71]. It further exploits data parallelism within windows with out-of-order bundle processing, as introduced by StreamBox [44].

4 KPA and Streaming Operations

This section first presents KPA data structures (§4.1) and primitives (§4.2). It then describes how KPAs and the primitives implement compound operators used by programmers (§4.2), and how StreamBox-HBM executes an entire pipeline while operating on KPAs (§4.3).

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 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. 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. 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, as discussed in Section 4.3.

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.

4.2 Streaming Operations

StreamBox-HBM implements the streaming primitives in Table 2, and the compound operators in Table 1. 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	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 $KPA(c) \rightarrow \{KPA_i(c)\}$	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.

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

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 [8]) as the key range of each output partition.

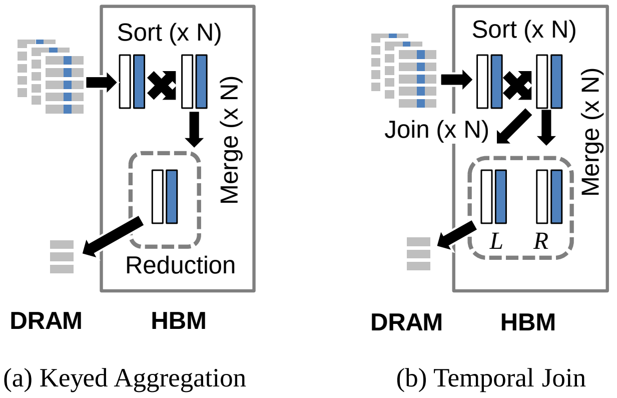


Figure 4. Declarative operators implemented atop KPAs

- **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 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 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.

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 [68] in Figure 1a to show pipeline execution. We omit *Projection*, since StreamBox-HBM stores results in DRAM. Figure 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 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.

5 Dynamically Managing Hybrid Memory

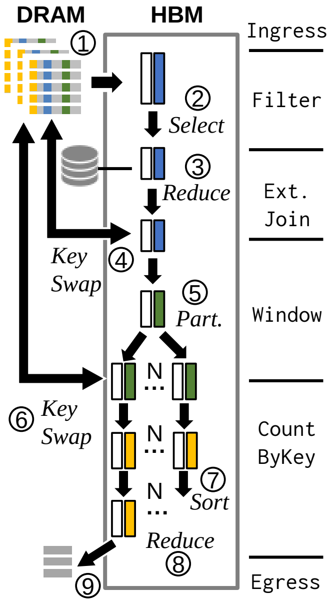


Figure 5. Pipeline execution on KPAs for YSB [68]. Declarative operators shown on right.

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.

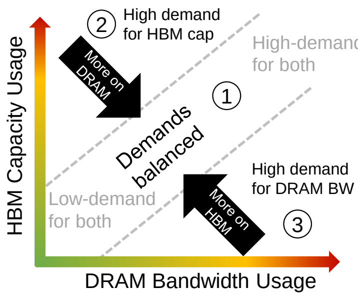
1. *Balancing demand.* StreamBox-HBM balances the aggregated demand for limited HBM capacity and DRAM bandwidth to prevent either from becoming a bottleneck.

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 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.

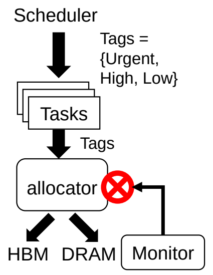
StreamBox-HBM monitors HBM capacity and DRAM bandwidth and trades them off dynamically. For individual KPA allocations, StreamBox-HBM further considers the critical path. StreamBox-HBM does not migrate existing KPAs, which are ephemeral, unlike other software systems for hybrid memory [61, 63, 72].

Dynamically Balancing Memory Demand Figure 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 oversubscribing 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



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



(b) Passing scheduling and resource knowledge to allocation

Figure 6. StreamBox-HBM dynamically manages hybrid memory

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 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.

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 [2]. 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.

6 Implementation and Methodology

We implement StreamBox-HBM in C++ atop StreamBox, an open-source research analytics engine [27, 44]. 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 [49].

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 [15, 16, 60]. YSB processes input records with seven columns, for which we use numerical values rather than JSON strings. Figure 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 [34]. 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 [68]. For Power Grid, we replay the input data from the benchmark [34]. 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 [6].

Hardware platform We implement StreamBox-HBM on KNL [33], 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 summarizes the KNL hardware and a 56-core Intel Xeon server (X56) used in evaluation for comparisons.

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			

Table 3. KNL and Xeon Hardware used in evaluation

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 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 [28]. With ZeroMQ ingestion, the engine copies incoming records from network messages and creates record bundles in DRAM.

7 Evaluation

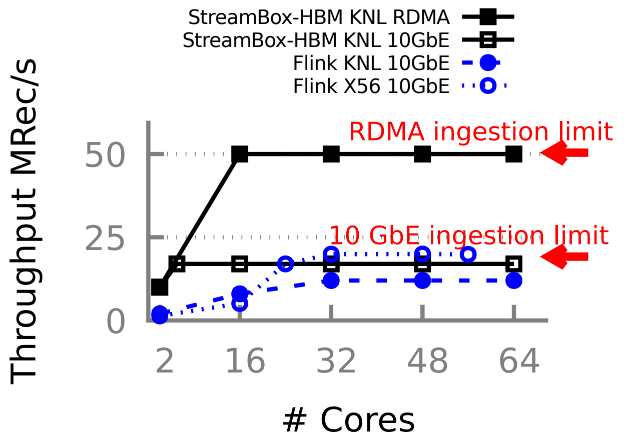
We first show StreamBox-HBM outperforms Apache Flink [12] on YSB. We then evaluate StreamBox-HBM on the other benchmarks, where it achieves high throughput by exploiting high memory bandwidth. We demonstrate that the key design features, KPA and dynamically balancing memory and performance demands, are essentially to achieving high throughput.

7.1 Comparing to Existing Engines

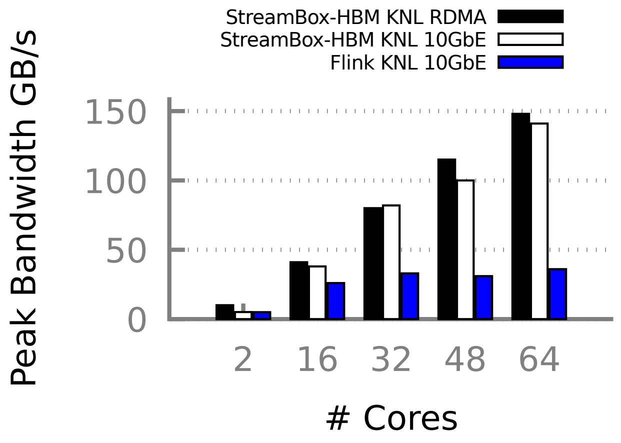
Comparing to Flink on YSB We compare to Apache Flink (1.4.0) [12], a popular stream analytics engine known for its good single-node performance on the YSB benchmark described in Section 6. To compare fairly, we configure the systems as follows. (1) Both StreamBox-HBM and Flink ingest data using ZeroMQ 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 because Flink targets such systems. We set the same target egress delay (1 second) for both engines.

Figure 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 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 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 [20, 44, 48, 57, 59, 71]. 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 [36] reports 30 M records/sec on a benchmark similar to Windowed Average, which is 4× lower than StreamBox-HBM as shown in Section 7.2. On a 24-core Xeon server, which has much higher core frequency than KNL, Tersecades [49], a highly optimized version of Trill [13], 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.



(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

Figure 7. StreamBox-HBM achieves much higher throughput and memory bandwidth usage than Flink, quickly saturating IO hardware. Legend format: "Engine Machine IO". Benchmark: YSB [68]

7.2 Throughput and Bandwidth

We use nine benchmarks and experimental setup described in Section 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 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

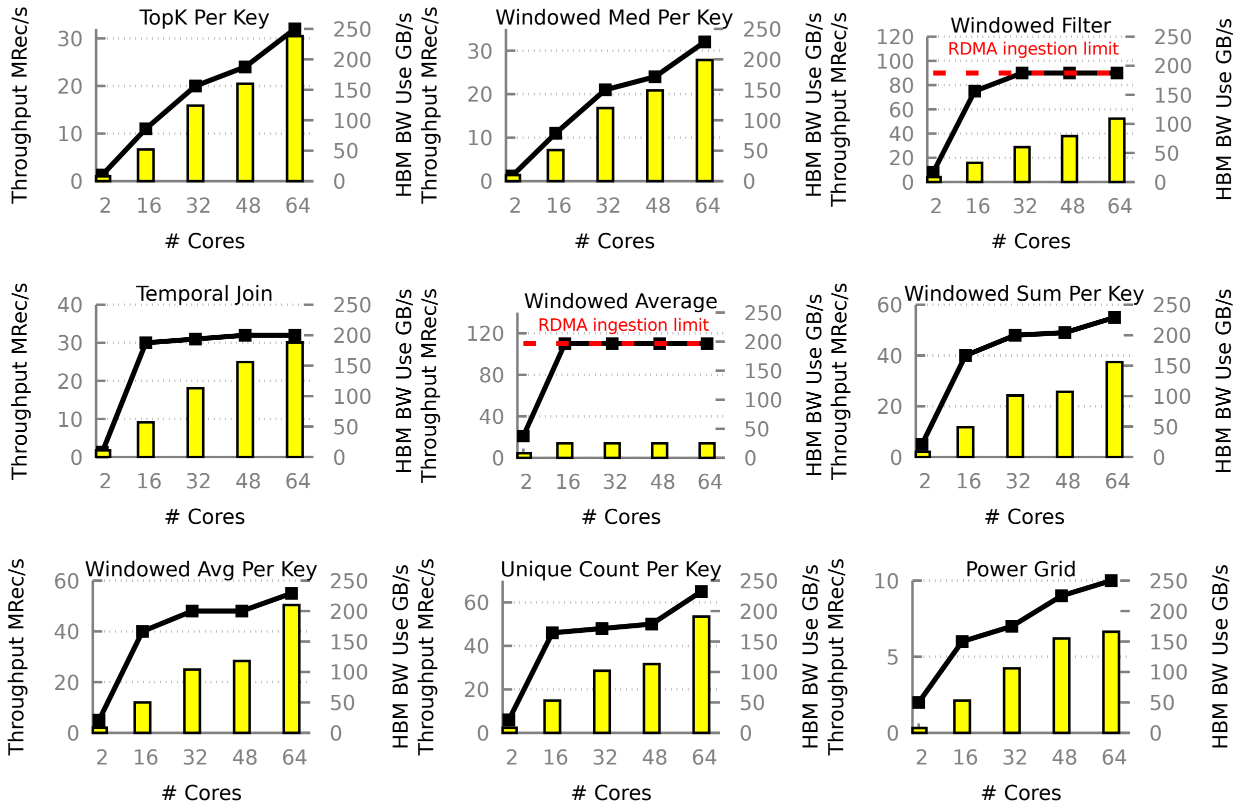


Figure 8. StreamBox-HBM's throughput (as lines, y-axis on left) and peak bandwidth utilization of HBM (as columns, y-axis on right) under 1-second target output delay. StreamBox-HBM shows good throughput and high memory bandwidth usage

achieve their maximum throughput 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.

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 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, 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 [44] with sequential algorithms on hardware-managed hybrid memory. Figure 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

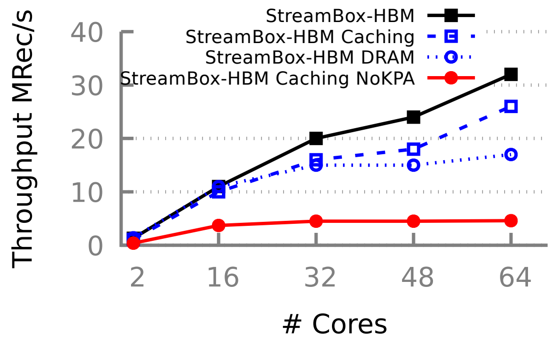
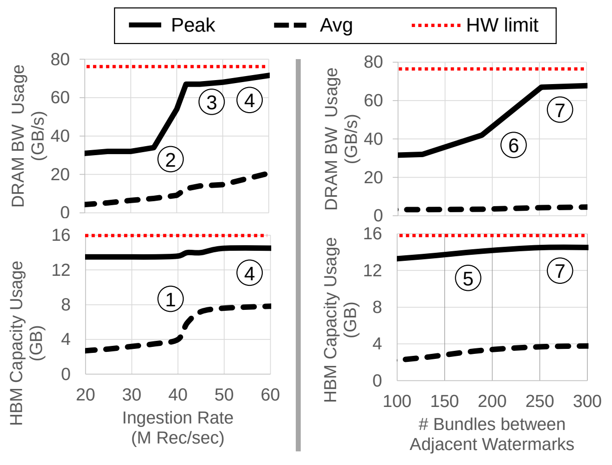


Figure 9. StreamBox-HBM outperforms alternative implementations, showing the efficacy of KPA and its management of hybrid memory. Benchmark: TopK Per Key



(a) Increasing ingestion rate (b) Delaying watermark arrival

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

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 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.

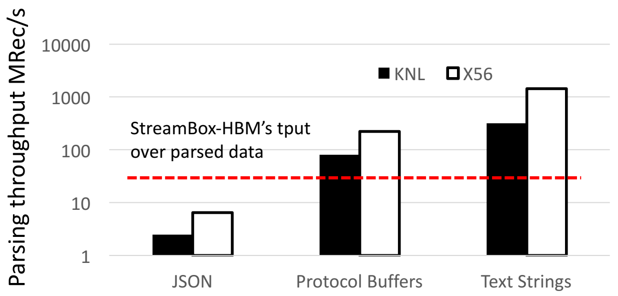


Figure 11. Parsing at the ingestion shows varying impacts on the system throughput. All cores on KNL and X56 are in use. Parsers: RapidJSON [69], Protocol Buffers (v3.6.0) [23], and text strings to uint64 [30]. Benchmark: YSB

Balancing memory demands To show how StreamBox-HBM balances hybrid memory demands dynamically, we increase data ingress rates to increase memory usage. Figure 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 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.

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 [49]. Yet, some streaming systems may ingest encoded data, 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) to see if the parsing throughputs can keep up with StreamBox-HBM's throughput on YSB.

As shown in Figure 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.

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 [44]. SABER accelerates streaming operators using multicore CPU and GPU [36]. Other work uses FPGA for stream processing [25]. No prior work, however, optimizes stream analytics for hybrid memories. StreamBox-HBM complements prior work that addresses diverse needs in distributed stream processing [4, 42, 45, 52, 59, 71]. They address issues such as fault tolerance [42, 52, 71], programming models [45], and adaptability [53, 60]. 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 [47]. Many relational databases store records in columnar format [10, 37, 54, 58] or use an in-memory index [39] 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 [13]. 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 [55]. Recent work optimized quick sort [11], hash joins [14], scientific workloads [40, 50], and machine learning [70] for KNL's HBM, but not streaming analytics. Beyond KNL, Mondrian [18] 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 [19]. Thermostat transparently migrates memory pages between DRAM and NVM while considering page granularity and performance [3]. CoMerge makes concurrent applications share heterogeneous memory tiers based on their potential benefit from fast memory tiers [17]. Tools such as ProfDP measure performance sensitivity of data to memory location and accordingly assist programmers in data placement [64]. 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 [43, 65]. 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.

9 Conclusions

We present the first stream analytics engine that optimizes performance for hybrid HBM-DRAM memories. Our design addresses the limited capacity of HBM and HBM's need for sequential-access and high parallelism. Our system design includes (i) novel dynamic key / record pointer extraction into KPAs that minimizes the use of precious HBM capacity, (ii) sequential grouping algorithms on KPAs to balance limited capacity while exploiting high bandwidth; and (iii) a runtime that manages parallelism and KPA placement in hybrid memories. StreamBox-HBM achieves 110 M records/second on a 64 core KNL machine. It outperforms engines without KPA and with sequential-access algorithms by 7× and engines with random-access algorithms by an order of magnitude. We find that for stream analytics, software better manages hybrid memories than hardware.

Acknowledgments

For this project: the authors affiliated with Purdue ECE were supported in part by NSF Award 1718702, NSF Award 1619075, and a Google Faculty Award. The authors thank the anonymous reviewers for their feedback. The authors thank Prof. Yiying Zhang for her technical suggestion and support. Heejin Park and Shuang Zhai contributed discussion on an early version of the paper.

References

[1] Apache Beam. https://beam.apache.org/.

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

[3] Agarwal, N., and Wenisch, T. F. 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* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 631–644.

[4] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., 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 8*, 12 (2015), 1792–1803.

[5] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., 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 8*, 12 (2015), 1792–1803.

[6] Albutiu, M.-C., Kemper, A., and Neumann, T. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow. 5*, 10 (June 2012), 1064–1075.

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

[8] Arasu, A., Babu, S., and Widom, J. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal 15*, 2 (June 2006), 121–142.

[9] Balkesen, C., Alonso, G., Teubner, J., and Özsu, M. T. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow. 7*, 1 (Sept. 2013), 85–96.

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

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

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

[13] Chandramouli, B., Goldstein, J., Barnett, M., DeLine, R., Fisher, D., Platt, J. C., Terwilliger, J. F., and Wernsing, J. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment 8*, 4 (2014), 401–412.

[14] Cheng, X., He, B., Du, X., and Lau, C. T. 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* (New York, NY, USA, 2017), CIKM '17, ACM, pp. 657–666.

[15] 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.

[16] 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.

[17] Doudali, T. D., and Gavrilovska, A. Comerge: Toward efficient data placement in shared heterogeneous memory systems. In *Proceedings of the International Symposium on Memory Systems* (New York, NY, USA, 2017), MEMSYS '17, ACM, pp. 251–261.

[18] Drumond, M., Daglis, A., Mirzadeh, N., Ustiugov, D., Picorel, J., Falsafi, B., Grot, B., and Pnevmatikatos, D. The mondrian data engine. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, pp. 639–651.

[19] Dulloor, S. R., Roy, A., Zhao, Z., Sundaram, N., Satish, N., Sankaran, R., Jackson, J., and Schwan, K. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 15:1–15:16.

[20] EsperTech. Esper. http://www.espertech.com/esper/, 2017.

[21] Facebook. Folly. https://github.com/facebook/folly#folly-facebook-open-source-library, 2017.

[22] Fluhr, E. J., Friedrich, J., Dreps, D., Zyuban, V., Still, G., Gonzalez, C., Hall, A., Hogenmiller, D., Malgioglio, F., Nett, R., Paredes, J., Pille, J., Plass, D., Puri, R., Restle, P., Shan, D., Stawiasz, K., Deniz, Z. T., Wendel, D., and Ziegler, M. 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.

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

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

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

[26] Hammarlund, P., Kumar, R., Osborne, R. B., Rajwar, R., Singhal, R., D'Sa, R., Chappell, R., Kaushik, S., Chennupaty, S., Jourdan, S., Gunther, S., Piazza, T., and Burton, T. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 2 (2014), 6–20.

[27] Hongyu Miao, Heejin Park, M. J. G. P. K. S. M., and Lin, F. X. Streambox code. https://engineering.purdue.edu/~xzl/xsel/p/streambox/index.html. Last accessed: July 25, 2018.

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

[29] 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.

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

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

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

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

[34] Jerzak, Z., and Ziekow, H. The debs 2014 grand challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems* (New York, NY, USA, 2014), DEBS '14, ACM, pp. 266–269.

[35] Kim, C., Kaldewey, T., Lee, V. W., Sedlar, E., Nguyen, A. D., Satish, N., Chhugani, J., Di Blas, A., and Dubey, P. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow. 2*, 2 (Aug. 2009), 1378–1389.

[36] Koliousis, A., Weidlich, M., Castro Fernandez, R., Wolf, A. L., Costa, P., and Pietzuch, P. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 555–569.

[37] Larson, P.-A., Clinciu, C., Fraser, C., Hanson, E. N., Mokhtar, M., Nowakiewicz, M., Papadimos, V., Price, S. L., Rangarajan, S., Rusanu, R., and Saubhasik, M. Enhancements to sql server column stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 1159–1168.

[38] Lee, D. U., Kim, K. W., Kim, K. W., Kim, H., Kim, J. Y., Park, Y. J., Kim, J. H., Kim, D. S., Park, H. B., Shin, J. W., Cho, J. H., Kwon, K. H., Kim, M. J., Lee, J., Park, K. W., Chung, B., and Hong, S. 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.

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

[40] Li, A., Liu, W., Kristensen, M. R. B., Vinter, B., Wang, H., Hou, K., Marquez, A., and Song, S. L. 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* (New York, NY, USA, 2017), SC '17, ACM, pp. 26:1–26:14.

[41] Li, J., Tufte, K., Shkapenyuk, V., Papadimos, V., Johnson, T., and Maier, D. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment 1*, 1 (2008), 274–288.

[42] Lin, W., Qian, Z., Xu, J., Yang, S., Zhou, J., and Zhou, L. Streamscope: continuous reliable distributed processing of big data streams. In *Proc. of NSDI* (2016), pp. 439–454.

[43] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 133–148.

[44] Miao, H., Park, H., Jeon, M., Pekhimenko, G., McKinley, K. S., and Lin, F. X. Streambox: Modern stream processing on a multicore machine. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference* (2017).

[45] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

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

[47] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., and Lomet, D. Alphasort: A risc machine sort. *SIGMOD Rec. 23*, 2 (May 1994), 233–242.

[48] OracleÂő. Stream explorer. http://bit.ly/1L6tKz3, 2017.

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

[50] Peng, I. B., Gioiosa, R., Kestor, G., Cicotti, P., Laure, E., and Markidis, S. 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.

[51] Polychroniou, O., Raghavan, A., and Ross, K. A. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1493–1508.

[52] Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., and Zhang, Z. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 1–14.

[53] Rajadurai, S., Bosboom, J., Wong, W.-F., and Amarasinghe, S. 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* (New York, NY, USA, 2018), ASPLOS '18, ACM, pp. 98–112.

[54] Raman, V., Attaluri, G., Barber, R., Chainani, N., Kalmuk, D., KulandaiSamy, V., Leenstra, J., Lightstone, S., Liu, S., Lohman, G. M., Malkemus, T., Mueller, R., Pandis, I., Schiefer, B., Sharpe, D., Sidle, R., Storm, A., and Zhang, L. Db2 with blu acceleration: So much more than just a column store. *Proc. VLDB Endow. 6*, 11 (Aug. 2013), 1080–1091.

[55] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

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

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

[58] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., and Zdonik, S. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases* (2005), VLDB '05, VLDB Endowment, pp. 553–564.

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

[60] Venkataraman, S., Panda, A., Ousterhout, K., Armbrust, M., Ghodsi, A., Franklin, M. J., Recht, B., and Stoica, I. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 374–389.

[61] Wang, C., Coa, T., Zigman, J., Lv, F., Zhang, Y., and Feng, X. Efficient management for hybrid memory in managed language runtime. In *IFIP International Conference on Network and Parallelly Computing (NPC)* (2016).

[62] Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., Zheng, C., Lu, G., Zhan, K., Li, X., and Qiu, B. 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.

[63] Wei, W., Jiang, D., McKee, S. A., Xiong, J., and Chen, M. Exploiting program semantics to place data in hybrid memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)* (2015).

[64] Wen, S., Cherkasova, L., Lin, F. X., and Liu, X. Profdp: A lightweight profiler to guide data placement in heterogeneous memory systems. In *Proceedings of the 32th ACM on International Conference on Supercomputing* (New York, NY, USA, 2018), ICS '18, ACM.

[65] Xia, F., Jiang, D., Xiong, J., and Sun, N. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 349–362.

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

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

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

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

[70] You, Y., Buluç, A., and Demmel, J. Scaling deep learning on gpu and knights landing clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2017), SC '17, ACM, pp. 9:1–9:12.

[71] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 423–438.

[72] Zhang, W., and Li, T. 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).