UNIVERSITY OF CALIFORNIA
RIVERSIDE

# On the Acceleration of Database Primitives on FPGAs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Bashar Romanous

September 2021

Dissertation Committee:

Professor Walid Najjar, Chairperson
Professor Nael Abu-Ghazaleh
Professor Vassilis Tsotras
Professor Daniel Wong

## Acknowledgments

Obtaining my doctorate degree at UCR has been one of the greatest experiences I had in my life. In addition to all the invaluable knowledge and experience that I gained, the personal and professional development as well as the precious friendships that I formed are things that I will always look fondly upon.

I am very grateful to my advisor, Professor Walid Najjar, for all of his guidance and advice. He taught me how to conduct research, overcome challenges, and present my work. His constant motivation and care have gotten me through the most difficult times, especially during the pandemic. Words cannot express my forever gratitude to you.

I would like to thank every member of my committee: Professor Vassilis Tsotras, especially for all of his inputs to the VLDB journal. I would like to thank Professor Nael Abu-Ghazaleh, and Professor Daniel Wong for all of their input, advice, and support of my research. I would also like to thank Professor Yan Gu for his contributions to the work on sample sort and Professor Evangelos Papalexakis for his inputs and contributions in my research.

I would like to thank all my friends. Nothing that I can write here can express how much your friendship means to me. We shared both the good and difficult times together, and I will forever cherish those memories. It has been my pleasure to share this journey with you. Thank you: Leen Kawas, Ignacio de Castro Perez, Devashree Tripathy, AmirAli Abdolrashidi, Jason Ott, Saheli Ghosh, and Ravdeep Pasricha. Also, I would like to acknowledge and thank my friends who were as my peer mentors as well: Skyler Windh, Jose Rodriguez Borbon, and Prerna Budhkar for teaching me their knowledge and experience in FPGA prototyping and research. To all of you, I will always cherish the beautiful memories and conversations we shared. I would also like to thank all my colleagues in the Embedded Systems lab.

Finally, this dissertation contains parts of two of my previously published works from my research. The first work explores an in-memory accelerator for hash-join and group-by aggregation operators using hardware multithreading techniques on FPGAs and was published in The VLDB Journal. It's content, with the exclusion of Section 3 and references to hash-join, appears in Chapter 3. Background & related work appear in Chapters 1 & 2 respectively. I would like to thank the reviewers and publishers for considering the work as well as my co-authors for their contributions. The full citation and author list is as follows:

Bashar Romanous, Skyler Windh, Ildar Absalyamov, Prerna Budhkar, Robert Halstead, Walid Najjar, and Vassilis Tsotras. 2021. Efficient local locking for massively multithreaded in-memory hash-based operators. VLDB J. 30, 3 (May 2021), 333–359.

The second work introduces HARS (Hardware Accelerated Radix Sort), a novel parallel implementation of radix sort on FPGAs that leverages the on-chip memory. The published material that appears in IEEE FCCM conference is a short summary of this work. An expansion of the published work is presented in Chapter 4. I would like to thank the reviewers and publishers for considering the work as well as my co-authors for their contributions. The full citation and author list is:

Bashar Romanous, Mohammadreza Rezvani, Junjie Huang, Daniel Wong, Evangelos E. Papalexakis, Vassilis J. Tsotras, Walid Najjar. 2020. High-Performance Parallel Radix Sort on FPGA. In 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 224–224.

To my parents who their sacrifices and hard work made me who I am.

To my dearest friends who are my family.

This work is dedicated to you.

ABSTRACT OF THE DISSERTATION

On the Acceleration of Database Primitives on FPGAs

by

Bashar Romanous

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2021
Professor Walid Najjar, Chairperson

The decreasing cost of DRAM has made possible and grown the use of in-memory databases. However, memory latency has not kept up with increasing processing speed and has become the limiting performance factor. Large cache hierarchies are effective in mitigating this large memory latency for regular applications that exhibit spatial and/or temporal locality. Database operations by their very nature, rely extensively on indirect addressing of sparse data and hence cannot benefit from large cache memories.

This dissertation describes the usage of filaments as an approach that focuses on latency masking rather than latency mitigation. Filaments are lightweight hardware threads that can issue multiple outstanding memory requests then switch to waiting state and yield control to other filaments that have their data ready. Filaments are implemented and evaluated to accelerate to hash-join and hash-based group-by aggregation on FPGAs. Content Addressable Memories (CAMs) are used as a synchronization cache that enables processor-side locking.

Sorting is an essential component in a wide variety of applications including databases, data analytics, and graph processing. While comparison-based and distribution-based sorting algorithms are the two common methods of sorting, comparison-based sorting algorithms are the most common to be accelerated on FPGAs. As the technology evolves and new hardware architectures and platforms are introduced a re-evaluation of accelerating distribution-based sorting algorithms, such as Radix sort, using FPGAs is justified.

This dissertation describes a novel parallel in-memory Hardware Accelerated Radix Sort (HARS), that does not rely on sorting networks and avoids the performance limiting merge steps. The scalability of datasets is not restricted by the available on-chip memory and does provide constant throughput.

Sorting very large datasets efficiently using limited local memory is a challenging problem. The common method of sorting a large dataset is to divide it into smaller sub-arrays that can fit on local memory, sort the sub-arrays and then merge the resulting sub-arrays until the entire dataset is sorted. The merge steps become memory and performance bottlenecks as the size of merged sub-arrays grow to exceed the local memory capacity.

This dissertation explores a parallel FPGA implementation of sample sort, a cache-oblivious sorting algorithm that enables sorting very large datasets using local memory. A set of pivots is used to distribute keys from sorted sub-arrays into buckets that can fit on local memory. Sorting the buckets produces the completely sorted set. All steps of sample sort can be performed entirely using FPGAs local memory. Thus overcoming a limitation of the currently available sorting routines on FPGAs.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Introduction

Fast analytic queries over large collections of customer data is a key workload for any modern business. As the cost of DRAM has continued to decrease, fairly large datasets can now be stored and processed entirely in memory. In recent years many database vendors have sprung up offering their own in-memory database solutions to speed up processing (MemSQL [95], SQL Server Hekaton [46], SAP HANA [48], Oracle TimesTen [85], IBM DB2 BLU [21]). The main factor influencing in-memory processing performance is memory bandwidth. Despite the progress made in multi-core architectures, the major performance limitations come from the memory latency (known as the *memory wall*), that restricts the scalability of such memory-bounded algorithms. A memory access can stall instruction execution for hundreds of CPU cycles.

The most common solution to the memory latency problem is the use of extensive *cache hierarchies*. This approach mitigates memory latency by relying on data and program instructions localities (spatial and temporal). Unfortunately, such solution does not come for free as cache hierarchies can take up to 80% of a typical processor die area, thus limiting the number of cores that can be accommodated on a single chip and also contributing to energy consumption through leakage current.

Moreover, there are many *irregular applications* that do not exhibit such localities [31]. As a result, cache hierarchies do not provide an effective solution for their memory accesses [35, 128]. In particular, irregular applications can be characterized by at least one of the following patterns: 1) Irregular control-flow which breaks the program locality. This is caused by branches in the code that invalidate pre-fetched instructions. 2) Irregular data-flow where indirection in the memory access patterns breaks the data locality and hence causes cache misses. Some database operators, such as *selection*, exhibit control flow irregularity, while others, like hash-based group-by aggregation can demonstrate both [47].

An alternative for dealing with the memory latency problem in irregular applications is offered by *hardware multithreaded execution* [83, 127]. This approach relies on the masking of memory latency by supporting multiple outstanding memory requests and switching to a ready but waiting thread when the currently executing thread encounters a long latency operation, such as a main memory access. Hardware multithreading was used in the SUN UltraSPARC architecture (for example, the UltraSPARC T5 [129]) where it can support eight threads per core and 16 cores per chip. However, tens of threads are not enough to hide memory latency. Instead we have recently advocated for *massive* hardware multithreading implemented on FPGAs (thereafter referred to as MultiThreaded Processor or MTP) that is able to support deeper pipelining, can maintain hundreds (instead of tens) of outstanding memory requests across four FPGAs and hence can *drastically* increase concurrency and therefore throughput [51, 61, 60].

In this dissertation we examine how to use our MTP approach to implement hash-based algorithms for the group-by aggregation [134, 11] operator. Group-by aggregation is a basic building blocks of relational query processor and various recent works have explored their implementation tailored to multi-core CPU architectures [40, 151]. A common component in hash-based group-by aggregation is to efficiently build a hash table, which is later used to return groups in the aggregated relation (potentially with appropriate aggregates). The hash-based nature of this algorithms incurs poor spatial locality, thus all the multi-core CPU approaches rely on vast caches to somehow alleviate latency penalty. In order to build a hash table, the MTP execution takes an alternative approach as it requires massive parallelism to compete with the CPU's order of magnitude faster clock frequency. In turn, that means many threads must be synchronized and managed locally on the FPGA. One could instead build a hash table in local on-chip memory (BRAM), as the BRAM's 1-cycle latency removes any need for synchronization. However, current FPGAs only have few MBs of local storage, which limits the hashed relation size only to few thousand records [62].

MTP multithreading was applied in [11] to implement the hash-based group-by aggregation. As group-by aggregation requires updating the hash table (to update the aggregate as a new record is added to a group), all writes to a hash table need to be synchronized. This dissertation, along with [134], extended the hash-based group-by aggregation operator accelerator [11] by exploring how Content Addressable Memories (CAMs) can be leveraged within the MTP multithreaded designs to enable processor-side locking by acting as a *synchronizing cache*. These CAMs enforce locks and merge threads together before they are written to memory thus enabling latency-masking of threads [133].

Locks used in [11] were implemented at the level of hash table buckets. These Coarse-Grained Locks (CGL) limited parallelism - especially in the case of skewed datasets. Fine-Grained Locks (FGL), proposed in [134], which are locks on the record level instead of the hash table buckets that reduce locking contention and improve parallelism and throughput.

This dissertation extends the work in [134] by using FGLs to further explore trade-offs between the sizes of synchronization and locks CAMs on performance and resources utilization. Furthermore, the design presented in this dissertation enables better utilization of memory channels, by using only a single memory channel per engine without any significant effect on throughput. An evaluation of throughput between the design presented in this dissertation and the group-by aggregation design from [11] demonstrates the performance benefits of this design.

Sorting is a key part in database applications (used in duplicate elimination, sort-merge joins and group-by aggregations). Sorting billions of records in a fast and energy efficient manner has become a key research challenge. In this dissertation, we explore sorting in-memory using a parallel version of Radix Sort to build a high-performance hardware accelerator, called HARS (Hardware Accelerated Radix Sort). Our design enables dividing the unsorted dataset among parallel engines without the need for a merge step. HARS is implemented using Micron's WX-2000 and SB-852 FPGA boards.

For very large sorting problems, the scalability the hardware accelerators in FPGA and ASICs is a significant problem due to the limited size of on-chip memory. The conventional solution is to divide the dataset into small sub-arrays, sort them and merge the resulting sub-arrays. The downside of this solution is that the merged sub-arrays size grows to becomes a memory bottleneck.

This dissertation explores a novel FPGA implementation of the parallel sample sort [25], a cache-oblivious sorting algorithm that provides a scalable sorting solution for very large datasets using local memory. After sorting small sub-arrays using local memory, a prefix-sum based on the a set of pivots that represent the dataset is used to distribute keys from these sorted sub-arrays into buckets. The resulting buckets are small enough to be sorted using local memory which the complete sorted set is produced. The sample sort accelerator is implemented entirely using Vits HLS and deployed on a Xilinx Alveo U280 Data Center accelerator card [146, 144].

## 1.2 Background

### 1.2.1 FPGA Heterogeneous Computing

FPGA architectures have evolved from their early stages into large logic arrays capable of concurrently executing multiple complex instructions. They have historically been used as off-chip accelerators where CPUs can offload compute intensive workloads, and read back the results. In recent years the FPGA has been trending closer and closer to the CPU. Xilinx currently offers a Zynq [149] line of chips that couples the FPGA's reconfigurable fabric with an ARM processor. While Intel introduced HARP [118] featuring Xeon+FPGA in a single platform targeting data analysis acceleration [59].

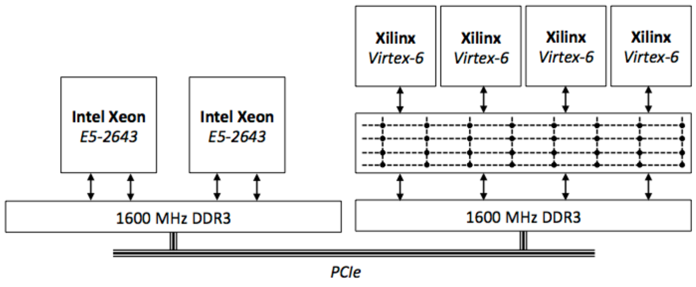
There has been various architectures where FPGA accesses memory directly, referred to Near-Data Processing (NDP). Some NDP platforms use FPGA to directly process data stored on none-volatile memory such as the FPGA-Accelerated flash storage proposed in [71] for sorting. Another type of NDP platforms focus on using the FPGA for data pre-processing. Thus, reducing performance bottlenecks caused by limited secondary storage and network bandwidth. Examples of such platforms are Netezza [52] used in data analytics and ExtraV framework [86] used in accelerating out-of-memory graph processing. Similar to Netezza, the Aqua platform, used by Amazon Redshift, a distributed and hardware accelerated cache, is another example of NDP technology [16, 44]. Nevertheless, it is currently still more common to see the FPGA connected with the CPU over a PCIe bus.

Microsoft Research incorporated multiple Stratix-V FPGAs into a 48 node server that was used to accelerate the Bing search engine [107]. Alpha Data announced a CAPI environment, which allows Xilinx All Programmable devices to connect with IBM Power8 architectures [13]. Multiple companies are offering FPGA platforms over PCIe, which have been actively used in the research community to show acceleration and power savings compared to CPUs and GPUs [60, 32].

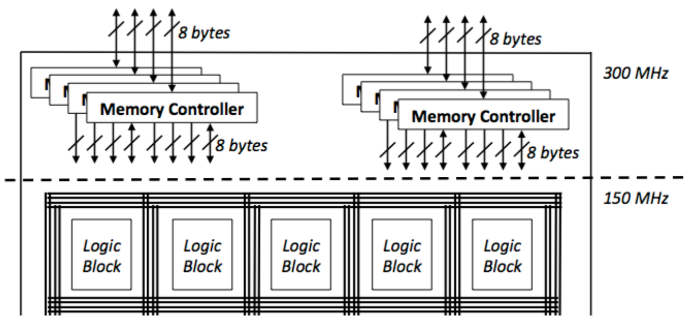
Different FPGA architectures are optimized for various use cases: HPC computations, database workloads or packet processing. The design proposed in this dissertation utilize only the off-chip memory interface, which makes them general enough to be ported between most currently available FPGA platforms. For simplicity we choose only one platform, the Micron (Convey) HC-2ex, to implement and run all our designs. The Convey architecture offers a shared global memory space between hardware and software, which eliminates any variability due to the memory architecture and allows us to do direct performance comparison against CPU and earlier FPGA implementations.

### 1.2.2 Convey Heterogeneous Computing Platforms

The Micron (Convey) HC-2ex is a heterogeneous platform that offers a shared global memory space between the CPU and FPGA regions. As shown in Figure 1.1a, the memory is



(a) The Micron (Convey) HC-2ex software and hardware regions.



(b) Micron (Convey) HC-2ex FPGA AE wrapper.

Figure 1.1: The Micron (Convey) HC-2ex architecture. Separation into software and hardware regions in shown in (a). In the hardware region each FPGA has 8 memory controllers, which are split into 16 channels for the FPGA's logic cells as shown in (b).

divided into regions connected through PCIe with portions closer to the CPU, and portions closer to the FPGAs. The software region has 2 Intel Xeon E5-2643 processors running at 3.3 GHz with a 10 MB L3 cache. In total the software region has 128 GB of 1600 MHz DDR3 memory. The system has a peak memory bandwidth of 51.2 GB/s.

The hardware region has four Xilinx Virtex6-760 FPGAs connected to the global memory through a full crossbar. Each FPGA has 8 64-bit memory controllers running at 300MHz (Figure 1.1b). The FPGA logic cells run in a separate 150 MHz clock domain to ease timing and connect to the memory controllers through 16 channels. These memory channels provide a highly parallel 8,192 simultaneous outstanding requests. The hardware region has 64 GB of 1600 MHz DDR3 RAM. Each FPGA has a peak memory bandwidth of 19.2 GB/s.

The Convey MX, used in our hash-join MTP implementation in [60], has the same hardware layout as the HC-2ex, however the MX has built-in support for atomic instructions directly in memory. The HC-2ex was used for the MTP group-by aggregation implementation in [11]. In this paper we use the HC-2ex as the common platform for both MTP hash-join and MTP group-by aggregation.

### 1.2.3 Latency-Masking Multithreading

The disparity between main memory latency and processing speed is one of the biggest challenges in computer design. On traditional CPUs and GPUs, it is addressed, by having very large cache hierarchies that mitigate the memory latency by exploiting spatial and temporal data localities. However, large classes of applications have very low levels of data locality, and hence do not benefit much from these massive cache hierarchies.

Many database operations fall into this category of applications. The objective of *latency masking multithreading* is to keep the processing unit busy while waiting for memory, thereby masking its latency. For database operations it is done by fetching and processing as many tuples, one per cycle, as the memory latency in cycles. In our model we associate a thread with the processing of a tuple. A thread is initiated when a tuple is read from memory and terminated when its processing is completed, whether results are materialized or not. In this paradigm, the processing of an item (a tuple in our case) is halted whenever a memory access is done and that *thread* is put in a *wait state* until the result of the memory access is returned. This means that the *state* of that thread is put in a queue waiting for memory as shown in Figure 1.2. When the data is returned from memory, in the case of a memory read, the thread moves to the next stage of its execution and the process is repeated for every memory access that the thread makes until its completion, i.e. until the processing of the tuple is completed. The parallelism, in this model, is equal to the number of *active threads*. A thread is active if it is either *executing* or *waiting*. In this model a thread goes through an execution pipeline consisting of successions of processing stages and waiting stages. Once this pipeline is full, the execution reaches a steady state and achieves the maximum throughput of one result per engine per cycle and memory latency is fully masked.

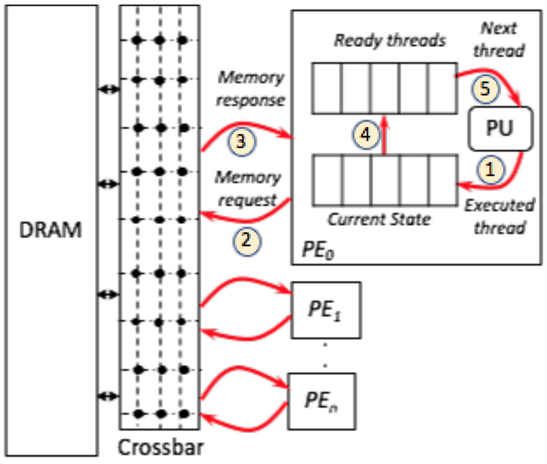


Figure 1.2: Components of a multithreaded implementation: numbers represent execution steps performed by a processing unit (PU) of each processing engine (PE). The design is scaled by adding as many PEs as possible.

Nevertheless, a synchronization mechanism must be provided when threads may write in memory. The main contribution of this dissertation is a processor-side synchronization mechanism relying on Content-Addressable Memories (CAMs) described in the next section. This mechanism is designed to synchronize all the threads *within* an engine.

On the Micron (Convey) HC-2ex, the average memory latency is ~100-200 cycles and memory accesses are fully pipelined, meaning that the processing unit can issue one memory request per cycle per memory channel up to the capacity of the memory buffer, which is ~500 memory requests. All memory requests, on the same channel, are returned in the order they were issued. There are three stages in the multithreaded execution that occur in the following order: 1) fill-up, 2) steady-state, and 3) drain-out. In the fill-up stage, the processing pipeline is being filled. Its duration is determined by the pipeline latency and the service time of each tuple. Conversely, the drain-out stage is when all the tuples have been read from memory and no new ones are inserted in the pipeline. When the number of tuples is much larger than the pipeline latency, the duration of the fill-up and drain-out stages is dwarfed by that of the steady-state, and hence the throughput is closer to the maximum achievable throughput.

In the experiments described in this dissertation, we implement a number ($N$) of engines on each FPGA. The total number of engines is therefore $4N$. Each engine is connected to $M$ memory channels on its FPGA. The data to be processed is partitioned equally among all the engines. All the engines operate concurrently and do not interact.

There is an inherent trade-off between the two parameters $N$ and $M$, as well as with the allocation of resources on the hardware to implement a given configuration. The choice of these two parameters is discussed in Section 3.2 for the MTP group-by aggregation.

The maximum achievable throughput in this model is determined by two factors: (1) the memory bandwidth, and (2) the number of active threads per engine. On the HC-2ex each FPGA has 16 memory channels and each channel can support one memory operation per cycle at 150 MHz. Therefore the whole system can achieve a peak of 64 memory operations per cycle, or 9.6 billion memory operations per second. The number of active threads within an engine is limited by the available resources on the FPGA. In an FPGA design every hardware item must be mapped, by a software tool, on some logic resources on the FPGA and items are connected together by routing wires between them. The length of the longest wire determines the clock frequency of the whole design. A design is said to meet timing if all the wires can be clocked at the target clock frequency. Selecting the best possible routing for every wire is an NP-complete problem. Very good heuristics take hours and some times days to achieve a routing that meets timing. The internal architecture of the Xilinx Virtex 6 FPGA is particularly challenging at meeting timing.

CAM-based synchronization is discussed in Section 3.2 for the MTP group-by aggregation. In both cases the size of the CAM was the main limitation on the number of active threads. The limitation on the CAM size comes from the impossibility of meeting timing for larger CAMs.

### 1.2.4 CAMs as synchronizing caches on FPGAs

A CAM (also known as an associative memory), is an array that can perform efficient entry-matching (i.e. answer membership queries). Its operation is the inverse of a Random Access Memory (RAM): when presented with a *search word* the CAM returns all the locations whose content matches that word. Each CAM bit consists of a flip-flop with a comparator matching it to the corresponding bit in the search word. The outputs of all the bit positions in a word are ANDed to generate the (mis)match for that word. The CAM's ability to perform a search in unit time comes at a high cost of area, energy and long clock cycle time (due to the long wires for the bit-wise AND and propagating the search word to all the entries). A CAM with $n$ entries where each entry is $w$

bits, stores $n$ words of $w$ bits, by construction. Each entry has a free bit and the next word to be inserted in the CAM is inserted in the first free entry and when a word is deleted from the CAM that entry is marked free. In this dissertation, depth or size of a CAM refer to the number of entries in the CAM.

As the number of entries in the CAM increases, the achievable clock frequency of the circuit drops. This limitation either restricts the size of the CAM or increases the number of cycles it takes to perform an update operation. Nonetheless, CAMs have proven to be very useful in domains such as networking (e.g. implementing an IP table in a network router). Recently we explored how CAMs can be used to accelerate the breadth first search algorithm [133]. These applications can usually tolerate long update latencies because update operations are infrequent.

In a streaming environment CAMs can maintain a cache of recently seen unique items and allow quick access to them without incurring long ready latency and stalling the pipeline. This fast cache look-up mechanism can also be used as a fine-grained address-based synchronization primitive, which avoids long latency trips to main memory and does not require special hardware synchronization primitives.

Consider the case when a CAM is assigned to guard a particular memory partition. It can be configured to hold the addresses of the values that need synchronized access. If all memory requests within a partition are first submitted to the CAM, before being routed to the memory, the accesses to identical addresses are serialized locally in the CAM. In this case a CAM entry serves as an exclusive lock, which gets released (flushed from the CAM) after the request(s) completion. In Section 3.4 we discuss how to use this approach for synchronization in the MTP group-by aggregation algorithm.

In our previous work [60] we have used atomic operations which were implemented using locks on individual memory locations, provided by the now discontinued Convey MX architecture [42]. Leveraging CAMs for synchronization of FPGA algorithms increases the portability of our design. Locking using generic CAMs means that all synchronization operations are now internal to the FPGA, and can be done on any architecture where an FPGA with a sufficient area has direct access to the memory. In addition, this design provides more selective fine-grained synchronization primitives in comparison to the Convey-MX, which places a lock on all FPGA-memory communication channels. We discuss resource trade-off for CAMs in Section 3.6. Work on previous FPGA implementations of CAMs appears in Section 2.2. An excellent description of CAMs and their applications can be found in [82].

# Chapter 2

# Related Work

## 2.1 Acceleration of Database operations

Commercial in-memory database systems include SAP HANA [48] which relies on multi-core CPUs, large main memory and caches as well as data compression. IBM BLU [21] attempts to keep data on the processor and its caches so as to reduce DRAM access as much as possible. MS SQL Hekaton [46] provides a lock-free data structure to provide high level of concurrency. Oracle's TimesTen [85] relies heavily on caches hierarchies. There are also academic research prototypes such as Peloton [104] which provides autonomous tuning for relational databases. GPU based solutions have also been proposed such as [63], while other are a hybrid of CPU and FPGA architectures have been proposed to mitigate long memory latencies [119].

Many recent works have considered the in-memory implementation of join and aggregation relational operators (hash- or sort-based). Sort-merge joins on modern CPUs were initially considered by Kim et al. [77]. This implementation explored the use of SIMD operations and hypothesized that sort-merge join performance will surpass the hash-based algorithms, given wider

SIMD registers. Subsequent work [12] implemented a NUMA-aware sort-merge algorithm that scaled almost linearly with the number of computing cores. This algorithm did not use any SIMD parallelism, but it was reported to be already faster than its hash join counterparts. Balkesen et al. [19] reconsidered the issue and found that hash joins still have an edge over sort-merge implementations even with the latest advance in width of SIMD registers and NUMA-aware algorithms. There has been a recent interest in tuning the join for specialized processing units such as Intel Xeon Phi [37]. PolyHJ [75] proposes a hybrid hash join paradigm that can dynamically execute different hash join models and tackle size skew. The proposed algorithm adapts behavior based on input relation and hardware characteristics.

In addition to joins, group-by aggregation operator, relying on multi-threaded architectures to boost its performance, was also extensively researched. One of the earliest works [40] explores different aggregation implementations on chip multiprocessors (CMPs) and concludes that performance largely depend on input characteristics like key cardinality, thus opting for adaptive strategy based on sampling.

Follow up work from the same authors [41] specifically explores the partitioning step of hash aggregation in the same CMP environment and, in line with [90], emphasizes the thread coordination as a key component of this step. The work by Ye et al. [151] considers both partitioning-based and non-partitioned aggregation implementations and proposes several hybrid approaches, which outperform previous implementations on Intel Nehalem architecture. In this work, we focus mainly on non-partitioned versions of algorithms. Wang et al. [131] describes novel NUMA-aware partitioned in-memory hash aggregation algorithm, which avoids cache coherency misses and minimizes locking costs. Finally, more recent work has shown the improvements of using novel CPU hardware. Cheng et al. [37] demonstrated a highly parallel in-memory join on the 64-core Intel Knight's Landing platform. Pohl et al. [106] have shown how HBM (high bandwidth memory) can be used as another layer in the storage hierarchy to improve performance. Hash and sort based aggregation are evaluated in [100]. The findings show that both approaches have the same complexity in terms of cache line transfers. An adaptive algorithmic framework based on sorting by hash value enables switching between hashing and sorting approaches during run-time based on a criterion of locality. The framework is cache-efficient and can be tuned depending on the hardware.

While the software community has examined both hash and sort-merge for join and aggregation operators the FPGA community has concentrated on sort-merge approaches. The reasons for this are twofold. Firstly, sorting and merging implementations are easily parallelized on FPGA architectures. For example, sorting networks like bitonic-merge [69] and odd-even sort [84] are well established designs for FPGAs; Casper et al. [32] developed a multi-FPGA sort-merge algorithm, while other works [112, 137] used sort-merge as part of a hardware database processing system. Secondly, building an in-memory hash table efficiently is non-trivial task because of the required synchronization.

An FPGA-accelerated implementation of group-by aggregation was first considered by Mueller et al. [98]. This work also utilized CAMs in the implementation of the aggregation operator, but in a very narrow scope, i.e. using CAMs to match an incoming tuple with the appropriate group. Hence the work continued long tradition of using CAMs to answer set-membership queries (previously explored in applications like click-fraud, online intrusion detection [20]). Our design also uses CAMs, but is different from previous approaches in two ways: (i) in addition to the key we store and update the aggregate value locally in the CAM, and (ii) we use CAMs as a synchronization primitive to resolve conflicts during updates.

Recently we used the MTP execution to accelerate the selection operator [30] by masking long memory latencies and managing thousands of threads concurrently without using any caches, as opposed to software CPU-based implementations, which require effective caching to limit memory requests. Using the MTP approach to implement a given database operator requires a design specific to the operator's characteristics. For example, the selection operator applies the query predicate to all the tuples in a relation. This operation is thus *partitionable*, meaning that each tuple could be processed independently of the others. Hence checking a predicate on a given tuple is an independent thread [30]; as a result, there is no need for inter-thread synchronization which is an important characteristic for both hash-join and group-by aggregation.

Kim et al. proposed BionicDB [78], an FPGA accelerator for OLTP workloads. Similar to our fine-grained locks, they used locks(implemented in BRAM) on the towers of jump-lists to improve throughput. Ma et al. [89] demonstrate a similar multithreaded FPGA model in the area of graph workloads.

## 2.2 Content Addressable Memories

A CAM is a memory where every bit is paired with a comparator allowing concurrent matching to a search word. Its ability to perform a search in unit time comes at a high cost of area, energy and long clock cycle time.

As the number of entries in the CAM increases, the achievable clock frequency of the circuit drops. This limitation either restricts the size of the CAM or increases the number of cycles it takes to perform a search or an update operation. Nonetheless, CAMs have proven to be very useful in domains such as networking (e.g. implementing an IP table in a network router). Recently we explored how CAMs can be used to accelerate the breadth first search algorithm [133].

In a streaming environment CAMs can maintain a cache of recently seen unique items and allow quick access to them without stalling the pipeline. This fast cache look-up mechanism can also be used as a fine-grained address-based synchronization primitive, which avoids long latency trips to main memory and does not require special hardware. Consider the case when a CAM is assigned to guard a particular memory partition. It can be configured to hold the addresses of the values that need synchronized access. If all memory requests within a partition are first submitted to the CAM, before being routed to the memory, the accesses to identical addresses are serialized locally in the CAM. In this case a CAM entry serves as an exclusive lock, which gets released (flushed from the CAM) after the request(s) completion. In [11] we discuss how to use this approach for synchronization in the multithreading group-by aggregation algorithm.

To the best of our knowledge all previous FPGA implementations relied on specialized platform features to provide synchronization primitives. In our previous work [60] we used atomic operations provided by the now discontinued Convey MX architecture [42]. Each word in memory maintains a locking bit that can be set by a specialized test-and-set memory instruction. Leveraging CAMs for synchronization increases the portability of our design by moving all synchronization operations to the FPGA. In addition, this design provides more selective fine-grained synchronization primitives in comparison to the Convey-MX, which places a lock on all FPGA-memory communication channels.

It was shown that implementing fully-associative matching logic for CAMs on both Altera and Xilinx FPGAs introduces a 60x space overhead compared to regular BRAMs [152]. Dhawan et al. [45] explored various designs of CAMs and introduced a trade-off between CAM size and update time.

## 2.3 Sorting Systems Hardware Accelerators

Sorting very large datasets is an extensively researched problem [67]. Research on sorting accelerators have targeted various platforms [58] such as FPGA [92, 91, 99, 71, 121, 81, 17], ASICs [124, 88, 49], GPGPUs [43, 15, 87, 33, 150], CPUs [28] and hybrid platforms [154, 55]. The most common used sorting algorithm has been merge sort followed by bitonic and even–odd merge sorting networks.

Near-storage sorting, proposed in [71], is achieved by having a flash expansion card connected directly to the FPGA board via two FMC ports. The proposed accelerator uses sorting networks and merge-sort units based on [121]. The implied throughput is approximately 14 million 16-byte records per second where the key is 80-bit. Unlike the work in [71], HARS targets in-memory sorting.

A parallel merge-tree proposed in [121] achieves a rate of one key/cycle/sequence.

FPGASort [81] considers different sorting architectures for high scalability algorithm. FIFO-merge and tree-merge have been utilized such that throughput scales with the number of memory channels. Partial run-time re-configuration is used to alternate between two merge units, thus enhancing resources utilization.

In [122] a bitonic merge network is used to alleviate the bottleneck imposed by serialization of merge sort in the final stages. A higher throughput is achieved. The sorted data is resident in on-chip memory and therefore size limited.

Sorting networks are, obviously, the most natural form of sorting architecture on FPGAs as surveyed in [99]. Even–odd and bitonic merge sorting networks require extensive hardware resources, hence, the importance of scalability [120].

In [23], eight sorting algorithms are implemented using HLS and compared in terms of execution time, standard deviation and resource utilization targeting the sorting of 4K keys, stored in on-chip memory, for real-time avionic systems applications.

Hybrid platforms have been explored to enable sorting of large scale datasets. A CPU-FPGA heterogeneous platform [154] proposed a “Streaming Merge” algorithm. Implementation targeted Intel HARP [118] which combines Intel Xeon E5-2600 processors with Intel (Altera) Stratix V FPGA. As FPGA reads and sorts blocks of data from shared global memory, the CPU merges the sorted blocks. Due to the fixed block size on FPGA, large datasets cause the merge sort on the CPU to become the performance bottleneck.

Chapter 3

# Enhanced Processor-Side Locking for In-Memory Operations

## 3.1 Introduction

Group-by aggregation is a memory intensive operator that is affecting the performance of relational databases. Hashing is a common approach used to implement this operator. Recent paradigm shifts in multi-core processor architectures have reinvigorated research into how the join and group-by aggregation operators can leverage these advances. However, the poor spatial locality of the hashing approach have hindered performance on multi-core processor architectures which rely on using large cache hierarchies for latency mitigation. Multithreaded architectures can better cope with poor spatial locality by masking memory latency with many outstanding requests. Nevertheless, the number of parallel threads, even in the most advanced multithreaded processors, such as UltraSPARC, is not enough to fully cover the main memory access latency. In this chapter we explore the hardware re-configurability of FPGAs to enable deeper execution pipelines that maintain hundreds (instead of tens) of outstanding memory requests across four FPGAs- drastically increasing concurrency and throughput.

In this chapter, we present an end-to-end in-memory accelerator for the group-by aggregation operator using FPGAs. The accelerator uses massive multithreading to mask long memory delays of traversing linked-list data structures, while concurrently managing hundreds of thread states across four FPGAs locally.

We also explore how Content Addressable Memories (CAMs) can be intermixed within our multithreaded designs to act as a *synchronizing cache*, which enforces locks and merges jobs together before they are written to memory.

The accelerator for the hash-based group-by aggregation operator demonstrates that leveraging CAMs achieves average speedup of 3.3x with a best case of 9.4x in terms of throughput over CPU implementations across five types of data distributions. Fast analytic queries over large collections of customer data is a key workload for any modern business. As the cost of DRAM has continued to decrease, fairly large datasets can now be stored and processed entirely in memory. In recent years many database vendors have sprung up offering their own in-memory database solutions to speed up processing (MemSQL [95], SQL Server Hekaton [46], SAP HANA [48], Oracle TimesTen [85], IBM DB2 BLU [21]). The main factor influencing in-memory processing performance is memory bandwidth. Despite the progress made in multi-core architectures, the major performance limitations come from the memory latency (known as the *memory wall*), that restricts the scalability of such memory-bounded algorithms. A memory access can stall instruction execution for hundreds of CPU cycles.

The most common solution to the memory latency problem is the use of extensive *cache hierarchies*. This approach mitigates memory latency by relying on data and program instructions localities (spatial and temporal). Unfortunately, such solution does not come for free as cache hierarchies can take up to 80% of a typical processor die area, thus limiting the number of cores that can be accommodated on a single chip and also contributing to energy consumption through leakage current.

Moreover, there are many *irregular applications* that do not exhibit such localities [31]. As a result, cache hierarchies do not provide an effective solution for their memory accesses [35, 128]. In particular, irregular applications can be characterized by at least one of the following patterns: 1) Irregular control-flow which breaks the program locality. This is caused by branches in the code that invalidate pre-fetched instructions. 2) Irregular data-flow where indirection in the memory access patterns breaks the data locality and hence causes cache misses. Some database operators, such as *selection*, exhibit control flow irregularity, while others, like *hash-join* and (hash-based) *group-by aggregation*, can demonstrate both [47].

An alternative for dealing with the memory latency problem in irregular applications is offered by *hardware multithreaded execution* [83, 127]. This approach relies on the masking of memory latency by supporting multiple outstanding memory requests and switching to a ready but waiting thread when the currently executing thread encounters a long latency operation, such as a main memory access. Hardware multithreading was used in the SUN UltraSPARC architecture (for example, the UltraSPARC T5 [129]) where it can support eight threads per core and 16 cores per chip. However, tens of threads are not enough to hide memory latency. Instead we have recently advocated for *massive* hardware multithreading implemented on FPGAs (thereafter referred to as

MultiThreaded Processor or MTP) that is able to support deeper pipelining, can maintain hundreds (instead of tens) of outstanding memory requests across four FPGAs and hence can *drastically* increase concurrency and therefore throughput [30, 51, 61, 60].

In this chapter we examine how to use our MTP approach to implement group-by aggregation [11] operator. Both operators are basic building blocks of relational query processor and various recent works have explored their implementation tailored to multi-core CPU architectures [18, 24, 77, 40, 41, 151]. A key component in group-by aggregation is to efficiently build a hash table (during the *build phase* of a join or grouping phase of the aggregation), which is later used to join with tuples from the second relation or to return groups in the aggregated relation (potentially with appropriate aggregates). The hash-based nature of these algorithms incurs poor spatial locality, thus all the multi-core CPU approaches rely on vast caches to somehow alleviate latency penalty. In order to build a hash table, the MTP execution takes an alternative approach as it requires massive parallelism to compete with the CPU's order of magnitude faster clock frequency. In turn, that means many threads must be synchronized and managed locally on the FPGA. One could instead build a hash table in local on-chip memory (BRAM), as the BRAM's 1-cycle latency removes any need for synchronization. However, current FPGAs only have few MBs of local storage, which limits the hashed relation size only to few thousand records [62].

This dissertation significantly extends our previous work, where we introduced the hash-based group-by aggregation operator accelerator [11]. Both works are extended by exploring how Content Addressable Memories (CAMs) can be leveraged within the MTP multithreaded designs to enable processor-side locking by acting as a *synchronizing cache*. These CAMs enforce locks and merge threads together before they are written to memory thus enabling latency-masking of threads [133].

In [11] we applied the MTP multithreading to implement the hash-based group-by aggregation. As group-by aggregation requires updating the hash table (to update the aggregate as a new record is added to a group), all writes to a hash table need to be synchronized. In [11] we used locks that were implemented at the level of hash table buckets. These Coarse-Grained Locks (CGL) limited parallelism - especially in the case of skewed datasets. Instead, our new design introduces Fine-Grained Locks (FGL), i.e. locks on the record level instead of the hash table buckets, thus reducing locking contention and improving parallelism and throughput. The FGL provides higher parallelism and throughput that is 1.6X to 2.1X times better than the work in [11]. Furthermore, to enable better utilization of memory channels, the new design uses only a single memory channel per engine without any significant effect on throughput. Finally, we implemented hash-based group-by aggregation on a common platform, the Micron (Convey) HC-2ex machine and rerun all experiments on this platform.

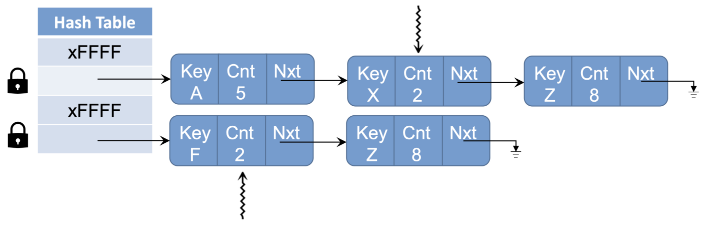
## 3.2 MTP Group-By Aggregation

In our group-by design, we assume the input relation fits in main memory but is too large to fit locally on the FPGA's memory. The aggregation engine is implemented with a single memory channel per engine (PE) *M=1* and *N=12* PEs. As the results will show, we found that condensing all engine requests onto a single channel and multiplexing them internally yields the highest utilization. This prevents stalls in one system component from stopping progress on multiple memory channels.

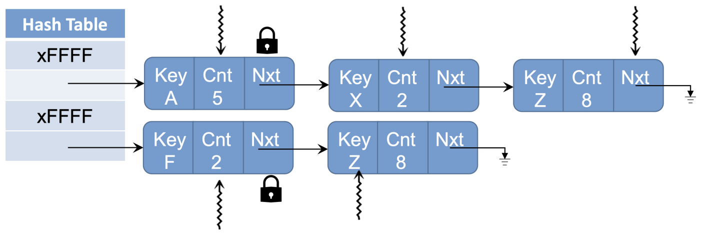
It also increases the inter-engine parallelism. The choice of parameters $M$, $N$ is further discussed in Section 3.6.

The mixed read-write nature of aggregation in conjunction with multiple outstanding memory requests requires explicit synchronization to ensure correctness. Atomic operations are one option, but this approach severely impacts the performance. Moreover, unlike the join operator, aggregated tuples may exhibit temporal locality.

We propose a novel multithreaded aggregation implementation based on CAMs [11] extended with fine-grained locks and efficient memory channel allocation. The design leverages explicit synchronization combined with the caching properties of the CAM. This fits perfectly in the context of group-by aggregation: Firstly, the latency of a single aggregation thread is hundreds of cycles, which means many threads can have identical keys. With a CAM we can merge these threads pre-aggregating the result locally on the FPGA and reduce the number of outstanding memory requests. This merging is achieved by leveraging caching properties of the CAM (allowing us to hold the aggregate value for a particular key): we call this the Filter CAM. It also allows us to alleviate skewed data distributions, where a subset of values appears as duplicate more often than the rest. Secondly, CAMs allow the FPGA to enforce locking on specific memory addresses, therefore decrease the granularity of the locks and boost the performance: we call this the Lock CAM. Each group-by aggregation engine uses one Filter and one Lock CAM. In the following description we assume a COUNT aggregation as an example. In the original aggregation design [11], our locks were implemented at the granularity of hash table buckets (Figure 3.1a). This guaranteed that only one thread was working on a list in the hash table at a time and it was free to modify the list as needed. With exclusive access to the list, the threads can perform node inserts in sorted order to



(a) Coarse-grained locks stop threads at the bucket level and prevent concurrent searching through the linked list.



(b) Fine-grained threads lock at the node level, increasing thread concurrency and only synchronize structural changes.

Figure 3.1: Coarse-grained vs. Fine-grained locking

improve the merge phase. However, such coarse-grained locking has a big impact on the parallelism the system is able to achieve. This is especially noticeable on skewed datasets where a majority of keys might map to the same bucket. All of those threads must stall and wait for the previous thread to finish. And the wait for each thread increases hundreds of cycles for each node added to the list. Each thread must pay this penalty even if it is only going to increment the count in a node and not modify the list structure.

## 3.3 Fine-Grained Locks

The first insight motivating the Fine-Grained Locks (FGL) design comes from the benefits of the top level Filter CAM. All new tuples that enter the aggregation start at the Filter CAM. If the key already exists, its count is incremented in the Filter CAM and the thread terminates. If the key does not exist in the Filter CAM and there is space, the key is added and a thread starts the hash table search. This construction guarantees that all threads searching the hash table are unique - they will never try to update the count in the same node because they all have different keys. We can take advantage of this design and move the lock lower to node pointers (Figure 3.1b). This enables synchronizing where it matters, when a thread wants to insert in the list and make a structural change.

### 3.3.1 Lock Free Reads

Since we only lock for structural changes (i.e. inserting a node), that means only writes need to be protected. This observation raises the question - *is it safe for threads to read past a lock?* Consider the possible situations of a thread progressing down a list (Figure 3.2). $T_1$ is a thread searching the linked list for a key $\beta$. $T_0$ is a thread inserting new node with key = $C$ and count = 2. Therefore, the next node field at node 1 is locked for the insertion of a pointer to the new node containing key = $C$ and count = 2. Since the link-list has key-values sorted, there are three possible outcomes of $T_1$ traversing the linked list:

1. $\beta > X$: There is no conflict between threads $T_0$ and $T_1$. Hence, the lock on the next node at Node 1 is irrelevant and it is safe for $T_1$ to proceed. $T_1$ will either find a node with a key equal to key $\beta$ or it will need to insert a new node with $\beta$ as the new key later in the list.

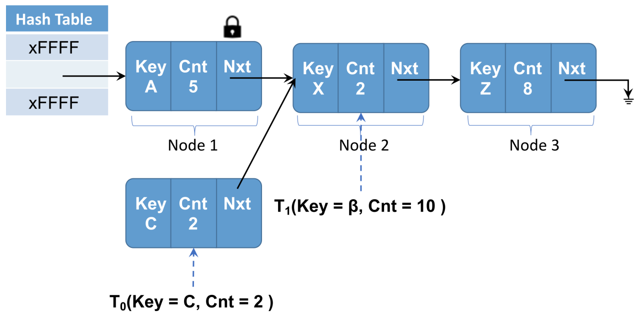


Figure 3.2: Demonstrating lock free reads. While new node is being inserted between Node 1 and Node 2, thread $T_1$ can traverses the liked list looking for $\beta$. If that key is found in the linked list, the count is updated, otherwise a new node is inserted.

2. $\beta = X$: In this case, $T_1$ has found its node, which is Node 2, and can update the count without synchronization.

3. $\beta < X$: Thread $T_1$ has progressed too far and must insert. However, insertion is gated by the lock on Node 1. $T_1$ will safely synchronize on that lock and will try again after the lock is free.

There are several benefits to using this design where reads are not locked. First, at no point does a thread need to stall until it needs to insert. If reads had to wait for the lock, these fine-grained locks could deteriorate to behavior like the coarse-grained lock - a thread blocks the start of the list and all work stops. However, the more important benefit comes from the behavior of the aggregation algorithm. For any given aggregation, the cardinality of the key can be much smaller than the size of the data table that is being scanned. Therefore, there will be a contentious period at the beginning of the execution where keys are getting inserted. However, once all keys have been seen once and inserted in their respective locations in the table, the rest of the execution

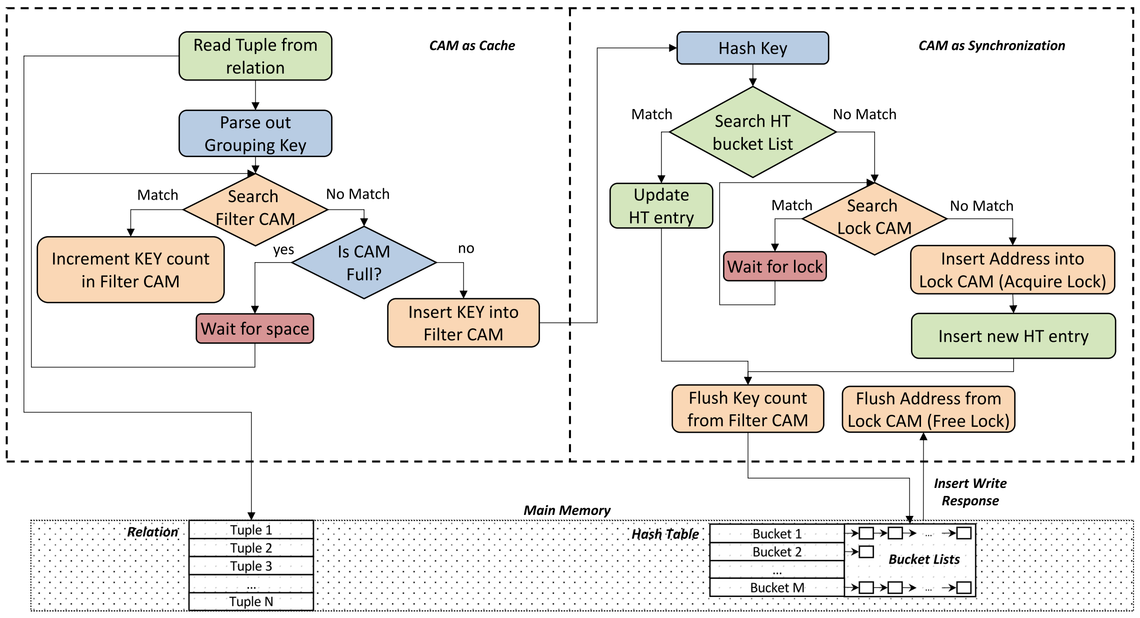


Figure 3.3: A state diagram for threads in the aggregation engine.

will proceed lock free. In the case where the cardinality is on the order of the number of items in the data-table, there may be no lock-free execution. However, this is still an improvement over coarse-grained locks since there will still be parallel inserts on the linked lists. As an example, consider the worst case dataset where all keys hash to the same bucket. With coarse-grained locks every item will be serialized for insertion by locking the head of the list to do the update. With fine-grained locks, the number of insert locations increases over time, decreasing contention and increasing the parallelism. If the keys are sorted, such that each ordered thread needs to append to the list, we lose the parallelism on insert, but we still gain in the parallel searching of the list for threads to find their insert location.

## 3.4 Aggregation Engine Workflow

Our design of an aggregation operation uses a custom hardware datapath called *aggregation engine*. Initially each tuple from the relation is streamed from memory, gets assigned to a separate MTP thread and starts its pipelined execution. Figure 3.3 shows the state diagram for a single thread inside the aggregation engine. The Filter CAM is used to merge threads with identical keys, hence reduces the memory request contention and minimizes the synchronization overhead. When there is a match in the Filter CAM, the thread will increment the key count in the Filter CAM. The thread that originally created this entry in the Filter CAM will update the HT with the new key count. However, due to hash collisions, the synchronization cannot be avoided completely; thus the Lock CAM is used to acquire locks on the hash table, ensuring its integrity. Each engine uses its own CAM for synchronization. As a result, values are aggregated in separate hash tables, which requires an extra merging phase at the end of the computation performed by the FPGA. Merging overhead grows as we increase the number of engines per FPGA, but it is an overhead that is amortized as the size of the dataset grows.

Table 3.1 shows an example of events and contents of Filter CAM, Lock CAM and main memory HashTable, while the input stream consists of 5 tuples with the following keys: $A$, $C$, $A$, $B$, $A$. Assume hash(A)=hash(C). Initially both CAMs are empty. The design assumes the COUNT aggregation function, thus the Filter CAM maintains an occurrence count of duplicate keys. However, other functions could be potentially applied. Note that operations updating the CAMs are performed immediately, whereas main memory HashTable accesses (e.g., search, entry update, entry insert) take hundreds of cycles to finish. For example, *Thread 1* sends a request to search value A in a hash table and gets response only at $Cycle_3$. Lock CAM maintains the locks for all addresses which are currently being modified. Notice that all threads only acquire locks after searching memory. Locks are only needed when creating a new node. Even though both *Thread 1* and *Thread 2* need to search the same bucket, they won't synchronize until after finding the list is empty and trying to add a new node. *Thread 1* finishes first and is able to get the lock and *Thread 2* finds it must wait in the next cycle. Once a thread completes, it invalidates the record in both CAMs and frees up resources for other threads. Threads, waiting for a place in a CAM, will continually cycle through a FIFO until the resource is available. Whenever there is a hit in the Lock CAM the thread waits until the lock is released, e.g. *Thread 2* resumes its work only at $Cycle_5$. *Thread 3* provides an example of early termination, because its value was locally aggregated in Filter CAM in $Cycle_3$. After $Cycle_5$ we see more concurrency as 3 threads are searching the list. *Thread 2* provides an example of fine-grained locking in $Cycle_7$. The thread gets to the end of the list and locks the next pointer of node $A$. At the same time, *Thread 5* is able to find node $A$ in the list and update its count without any locks. Finally, *Thread 4* is able to finish in $Cycle_{10}$ after a long memory request and waiting for a free cycle in the Lock CAM. This code can be adapted to fit other forms of aggregation operations such as SUM(), AVG(), MAX() and MIN(). In order to support MAX function instead of COUNT, the function in the Filter CAM needs to change from performing increments by 1 to computing MAX (current_max, new_value). Both functions would require reading the current value from memory (COUNT or MAX). While the COUNT always updates the count in memory, MAX may or may not update the value in memory. As for AVG(), the Filter CAM is still needed to count and then divide by number of elements. On the software side, an opcode can be used to activate a different function on the FPGA.

## 3.5 MTP Design Optimizations & Trade-offs

The main factor limiting performance of this memory-bounded problem is the efficient use of available memory bandwidth. In this chapter we use a Micron (Convey) HC-2ex machine, but our design is platform independent. In the HC-2ex the communication between the FPGA and main memory relies on the abstraction called *channel*. Each channel supports independent and concurrent read/write accesses to memory.

The original design of our aggregation engine required 4 memory channels: one for streaming the input tuples, one for accessing the in-memory hash table, and finally two channels for the bucket lists read/write operations. Since the Micron (Convey) HC-2ex has 16 memory channels, we replicated 4 engines ($\frac{16}{4}$) on a single FPGA thus leveraging inter-engine parallelism. Our original experiments showed that some memory channels were idle for almost 70% of the total execution time. Since the channels within an engine are statically assigned to perform different functions of the pipeline, back pressure from some components (e.g. thread waiting through CAM synchronization) introduces stalls and decreases the effective throughput.

In order to increase memory utilization we then *multiplexed* a pair of engines on the same set of memory channels, thus allowing the same channel to be used by two different engines. This means that the following engine operations: 1) Send and receive tuple request and response, 2) Read and write respective values to the hash table, 3) Read and write entries into respective bucket list, can run concurrently on two different engines. The multiplexed design increases the number of CAMs that could be placed on the FPGA, leading to further improvement in throughput. Unlike the original design[11], the new multiplexed engine uses 5 memory channels (adding an extra channel for accessing the in-memory hash table). This enabled 6 engines ($2 * \lfloor\frac{16}{5}\rfloor$) on a single FPGA.

In this latest design, to further improve the utilization of memory bandwidth, we have reduced the number of memory channels down to 1 per engine. For each engine, all requests for streaming in the tuples, accessing the hash table, and accessing linked lists are multiplexed internally. This construction enables up to 16 engines per FPGA, however, due to routing constraints we only implemented 12 engines. While multiplexing more requests over a given channel likely increases the latency of any given thread, it enables the engine to always have a request available to issue and can keep the engine in the steady state longer. If the aggregation datapath is working through requests but cannot take any more tuples from the stream, it is a waste of bandwidth to dedicate a channel to streaming tuples. As the percentage of execution time in the steady state increases relative to the fill-up and drain-out stages, the overall throughput of the system increases.

## 3.6 Experimental Results

The MTP aggregation implementation is compared in terms of overall throughput against the best multi-core approaches [40, 151] running on a single processor with 4 parallel threads. A summary of the various software aggregation algorithms follows, as well as a description of the datasets used in the experiments.

### 3.6.1 Software Implementations

In order to evaluate our MTP architecture we have implemented the following state-of-the-art multithreaded software aggregation algorithms: (i) Independent Tables[40], (ii) Shared Table [40], (iii) Hybrid Aggregation [40], (iv) Partition with Local Aggregation Table [151] and (v) Partition & Aggregate [151]. Here, (i) and (ii) are considered as non-partitioned approaches, while (iii) and (iv) are hybrid, and (v) is a partitioned approach.

- **Independent Tables** [40] is the approach most similar to our hardware implementation. The tuples are evenly split among separate software threads (without partitioning), and each thread aggregates result into its own hash table. Once the aggregation is complete all tables are merged together, which requites write synchronization.

- **Shared Table (with locking or atomic synchronization)** [40] splits the tuples evenly between threads, but all threads aggregate their results into a single hash table, hence no extra merge step is required. The algorithm could use different synchronization primitives: either pthread mutex implementation or Intel-specific hardware atomic instructions. Preliminary experiments showed that atomic primitives are significantly better on low key cardinalities, and don't have any difference from mutexes on medium and large cardinalities, so we choose atomics as a default synchronization primitive in all further experiments.

- **Hybrid Aggregation** [40] is a combination of two previous approaches. This algorithm allocates a small hash table for each thread. The size of the table is calculated based on the processor's L2 size to avoid cache misses. If the local table has enough space for a new value, or the value already exists in the table, that tuple is locally aggregated. Once the local table is filled and the next tuple requires a new slot, the oldest entry in the cached table will be spilled into larger shared table, residing in main memory, thus maintaining only "hot" data in L2 cache. Once aggregation is complete all small cached tables are merged into the large shared table. Merge step is synchronized as in *Independent Tables* case.

- **Partition & Aggregate** [151] (also known as count-then-move [41]) uses individual tables per thread, but before aggregation is performed the tuples are partitioned, in contrast to all aforementioned approaches. Separate partitioning step makes sure that all threads will work on non-overlapping values, hence aggregation could be done without any synchronization and the final tables are simply concatenated, rather than merged. As with the partitioned join implementations *radix clustering* algorithm is a backbone of this preliminary step.

- **PLAT (Partitioning with Local Aggregation Table)** [151] is a combination of two previous techniques. The algorithm takes advantage of the fact that we are performing an additional data scan, while doing a pre-processing step. While partitioning tuples into groups with mutually exclusive keys, each thread tries to aggregate values into its own small L2-resident table, as in *Hybrid Aggregation* approach. Values that do not fit into the small table are partitioned using *radix clustering* algorithm. Once pre-processing is done standard lock-free aggregation is applied. To finish, all tables which were produced during aggregation are concatenated together, while local tables are synchronously merged.

### 3.6.2 Dataset Description

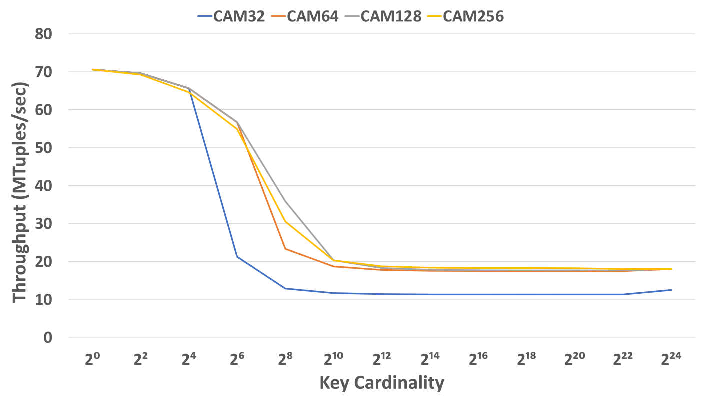
We use five datasets with various key distributions, namely: Uniform, Heavy Hitter, Moving Cluster [40], Self Similar and Zipf_0.5.

- In the **Uniform** dataset all key values are picked from the *uint64* key range with uniform probability. After that generated key/value pairs are randomly shuffled.

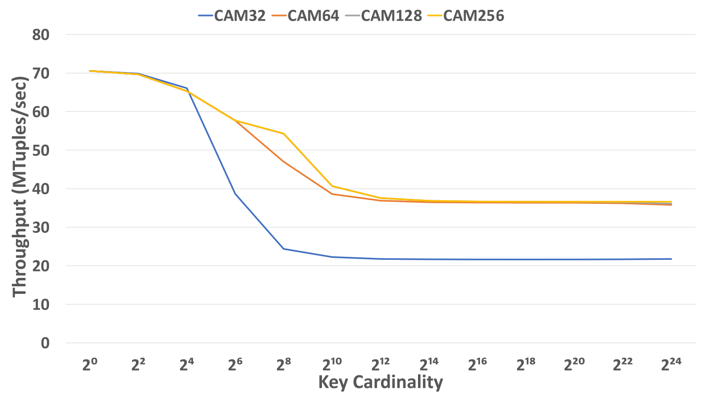
- A half of the tuples in the **Heavy Hitter** dataset [40] share the same a key value. The remaining key values are picked uniformly and evenly distributed throughout the the entire relation.

- In the **Moving Cluster** dataset [40] tuples are grouped into clusters depending on their key values. Lower key values are more likely to appear at the beginning of the relation, whereas tuples with higher key values are tend to appear at the end of the relation.
- **Self Similar** uses Pareto rule to model key distribution in a dataset: a single key value is shared by 20% of the tuples. Of the remaining 80% of tuples 20% of those share another key value. This process is repeated recursively to generate the relation. Tuples are randomly shuffled. The generation algorithm is described by Gray et al. [56].
- In the **Zipf** dataset key values follow the Zipf distribution with a skew coefficient of 0.5. The generation algorithm appears in aforementioned work[56].

Each dataset consists of several benchmarks with cardinalities ranging from $2^{10}$ to $2^{22}$ unique keys. The relation size in all of the experiments was 256 million tuples (in line with previous research [151]). Each dataset used the same 8-byte wide tuple format, which is commonly used for performance evaluation of in-memory query processing algorithms [19, 26, 24] and represents a popular column-wise storage format. The first 4 bytes of the tuple hold the unique primary key, while the rest is reserved for the grouping key. Since we are only interested in counting records with the same grouping keys, our tuples do not store any other information. However, none of the design choices prevent the use of "wide" tuples (i.e. containing fields other than primary and grouping keys). This could be easily supported by adding a key extraction component into the MTP design. Moreover experimenting with such "skinny" tuple format yields the best performance for software implementations, since it minimizes the cache capacity misses, which would decrease caching effectiveness otherwise.



(a) Uniform



(b) Heavy Hitter

Figure 3.4: Aggregation throughput of single engine for 256M tuples as the Filter CAM size is varied from 32 to 256

### 3.6.3 Effect of Filter CAM Size in MTP Group-By Aggregation

The throughput of a multithreaded engine is determined by the number of threads needed to fully mask latency. In this FGL engine, one of the key controls on the number of threads concurrently working is the size of the Filter CAM. Since every entry in the Filter CAM starts a thread searching the hash table for a node, we started by experimenting on the effect of the CAM size on throughput. Figure 3.4 shows the throughput as a function of the Filter CAM size for two of the data distributions, and illustrates the caching aspect of a CAM. The other distributions show similar behavior.

For the uniform distribution (Figure 3.4a) there is a sharp drop in throughput where cardinality grows larger than the Filter CAM size. As the cardinality increases, there will be little temporal locality in the tuple stream and pre-aggregation provides little help. CAM sizes of 64 or 128 provide similar throughput, especially for larger cardinality where hash table searching dominates.

For the heavy hitter distribution (Figure 3.4b) there is a similar drop in throughput where cardinality grows larger than the Filter CAM size. However, in this instance the Filter CAM size definitely affects the achievable throughput when hash searching dominates. A CAM size of 32 keeps the throughput similar to the uniform distribution. CAM sizes of 64 and above nearly double the throughput as the Filter CAM is able to exploit locality in the skewed data. As with all caches, there are diminishing returns for increasing the size. Maintaining locality for larger cardinalities requires significantly larger CAMs. Considering the diminishing returns of CAM sizes above 64, our current design uses a Filter CAM size of 128 to ease resource pressure.

### 3.6.4 Throughput Evaluation

Figures 3.5, 3.6, and 3.7 shows the throughput of group-by aggregation as the key cardinality is increased, obtained for various distributions. Throughput was measured for two MTP engine designs: multiplexed [11] and FGL, and five software implementations: two non-partitioned, two hybrid and one partitioned. Throughput for the skewed Heavy Hitter dataset Figure 3.6b resembles the results for Self Similar dataset Figure 3.5b, while the throughput for moderately skewed data Zipf_0.5 shown in Figure 3.7 is similar to the results obtained for Uniform dataset Figure 3.5a. Software implementations demonstrate the best performance on Moving cluster dataset Figure 3.6a due to the property of the data distribution: similar grouping keys appear in the input stream clustered together, increasing CPU-cache hit rates.

Despite all the differences in data distribution, the CPU aggregation performance is mainly determined by the dataset's key cardinality. When the number of unique keys is low, hash tables can fit into the CPU cache entirely. However, as the cardinality increases, cache misses start to hamper the throughput due to high latency memory round-trips. Software performance severely deteriorates at cardinalities higher than $2^{18}$ on all datasets for all algorithms. Another trend, which appears in all experiments, is that the Independent Tables approach yields the best result across all software algorithms. Nevertheless, that algorithm exhibits poor scalability, since the amount of memory needed for aggregation processing grows linearly with the number of parallel threads and the key cardinality. As the number of parallel threads increases, the amount of available memory could quickly become a bottleneck. We could also see that hybrid algorithms (PLAT and Hybrid Aggregation) outperform traditional partitioned (Partition & Aggregate) and non-partitioned (Shared Table) approaches by amortizing the cache miss cost and sustain a throughput around 400 MTuples/sec. This trend continues for cardinalities up to $2^{16}$, which marks the end of L3-cache residency. After that point the performance advantage of hybrid algorithms vanishes and drops below 100 MTuples/sec.

The throughput of the MTP designs also drops as the key cardinality increases, however, this effect is much less profound. Unlike the software throughput, this drop is explained by the overhead, introduced by the post-processing merge step.

The most striking aspect of the MTP throughput is that it is mostly constant across the range of cardinality dropping by a factor of two at the most while the software one drop by factors in the 100s. The results clearly show the benefits of the FGL MTP design over the multiplexed one. An important detail to note is that the FGL implementation is only using 12 engines, and thus only using 75% of the available memory bandwidth. Even with this handicap in bandwidth, FGL design is able to outperform the multiplexed design. The increased throughput comes from three main factors. First, reducing the number of channel allocations lets us use more engines and hence see increased inter-engine parallelism. Second, because we are multiplexing more requests over the same channel, this design is able to use the available bandwidth more efficiently. This better efficiency improves the latency-masking and increases throughput. Last, the fine-grained locks mean that we only need to do locking in the beginning of execution while the first nodes are being inserted. The remainder of the execution is lock free.

### 3.6.5 Trade-Offs

For a given workload, as the number of engines in the machine increases, the work done in each engine decreases and the time spent by this engine in the steady state, at maximal throughput, is reduced, hence the overall throughput is reduced as more time is spent in the fill-up and drain-out stages.

We varied the number of engines per FPGA from 1 to 12 (4-48 total engines), and tested it with Uniform key distribution on 256M tuples dataset. The results in Figure 3.8 show that the throughput is linear with the number of engines. As the number of engines grows higher, the bandwidth starts to saturate, causing the gain in throughput to decrease. The gain in throughput when going from 40 engines to 48 engines is less than that when going from 32 engines to 40 engines.

The comparison of the FGL throughput to that of the multiplexed design appears in Figures 3.5, 3.6 and 3.7. The following factors contribute to the decline in throughput:

1. The locking ratio increases with cardinality, taking up a much larger portion of the execution time.
2. FGL MTP design has 12 engines versus 6 in the multiplexed MTP design, which means each engine has less data to work on.
3. FGL MTP design has a smaller Lock CAM of depth 32 compared to 128 in the multiplexed MTP design. It was not possible to place and route 12 engines with the larger CAMs on the Xilinx Virtex6-760 FPGA.

We note that a newer FPGA could fit a larger CAM and 12 engines eliminating the drop in throughput.

**Discussion**

The performance benefits of the MTP approach come not from architecture-specific features of the FPGA devices, but from the massive multithreading that enables the MTP engine to better utilize the available memory bandwidth while masking latency. Figure 3.9 depicts the ratio of effective average memory bandwidth to peak theoretical memory bandwidth for the best software (Independent Tables) and the MTP implementations while varying the dataset sizes and key cardinalities. The MTP approach allows the FGL MTP to keep the ratio almost constant, irrespective of the dataset size or key cardinality. Since the FPGA's and CPU's memory bandwidth are 38.4 GB/s and 51.2 GB/s respectively then ratio 1 in Figure 3.9 corresponds to 38.4 GB/s on the HC-2ex and 51.2 GB/s on the CPU.

For the software implementation, at low cardinality the aggregated relation and hash table are mostly in the cache hierarchy and there are almost no memory accesses, hence the ratio approaches zero. The ratio peaks at around 0.5 for cardinality $2^{18}$, but drops significantly for higher key cardinalities.

The effective memory bandwidth of the CPU implementation tends to grow as the size of the relation increases (from 16M to 128M), whereas the MTP approach is less susceptible to data size variations. For average cardinalities, the FGL MTP implementation is almost 30 times higher.

For very large cardinalities, the FGL MTP implementation ratio drops due to the small Lock CAM as explained in Section 3.6.5. Yet, the ratio is still about 2.5 times higher on average than the software.

### 3.6.6 Fine-Grained Locks versus Coarse-Grained Locks

The performance benefits of FGL compared to the CGL [11] is shown in Figures 3.10a (for the Uniform key distribution dataset) and 3.10b (for the Heavy Hitter key distribution dataset). For this experiment, both FGL and CGL designs have the same number of engines and the same sizes of Filter and Lock CAMs. The throughput is normalized over the available bandwidth. The FGL design normalized throughput is 4 times higher than that of CGL design demonstrating the reduction of the locking overhead as discussed in Section 3.3.

### 3.6.7 Effects of the Merge Operation

Figure 3.11 shows aggregation throughput while the size of the datasets having Uniform key distribution is increased. The parallel MTP aggregation step has almost constant throughput of about 820 MTuples/sec, which drops on very high cardinalities due to the usage of a small Lock CAM as explained in 3.6.5. The effect of a small Lock CAM size is less pronounced on larger datasets as the engines spend more time in the steady state. The merge step introduces an overhead, however it comes at a fixed price. This cost depends solely on the key cardinality because aggregation reduces the initial input into a constant number of streams which should be merged. Hence, as the size of the relation grows the merge step overhead gets amortized.

### 3.6.8 FPGA Resources Utilization

Table 3.2 shows the resource utilization (registers, LUTs, and BRAMs used) for the two MTP aggregation designs (multiplexed, FGL) as the number of engines is scaled up. The biggest drivers of resource usage in these engines are the CAMs. The CAMs are the largest components in the engines and dictate size and timing constraints. It's interesting to note that the 8 engine FGL design is comparable to the previous design, showing that the increased complexity of the lower level locks is not too complex to implement in hardware. We were also able to save significantly in BRAM usage as well. The aggregation design uses only 62% of the available resources showing there is still room to incorporate other relational operations on the FPGA fabric.

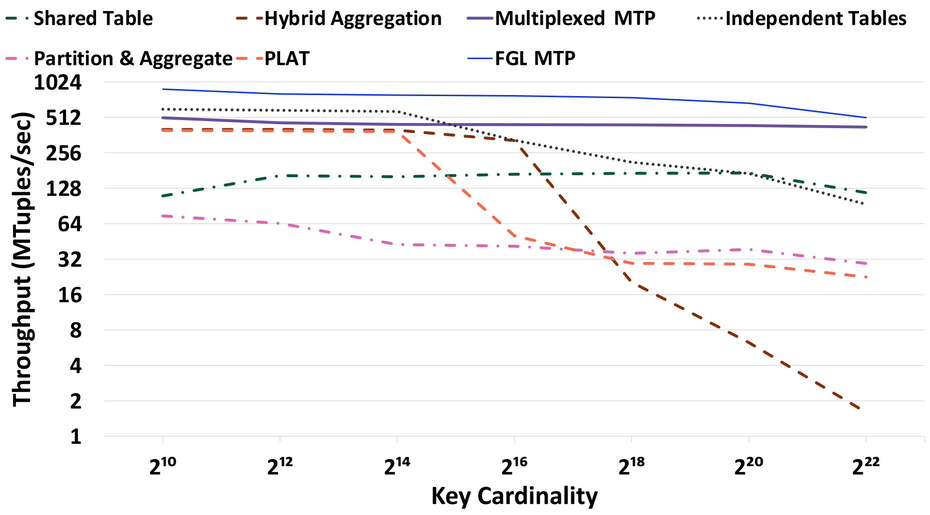
## 3.7 Conclusion

In this chapter, we implemented and evaluated hash-based group-by aggregation, using hardware multithreading techniques on FPGA hardware accelerators. The data structures are kept in global memory, which increases the access latency compared to on-chip BRAMs, but allows us to tackle much larger problem sizes. Multithreading allows the MTP design to mask the longer latency by issuing hundreds of threads across four FPGAs.

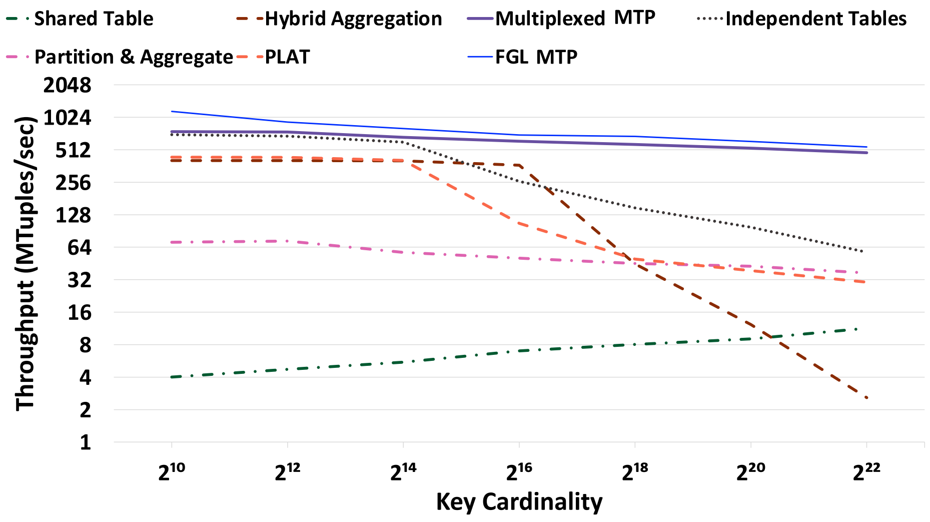
Multithreading techniques are used to implement a group-by aggregation function on the MTP design. Aggregation is a complex operation because threads can either update an existing node, or create a new node. We evaluate the FGL MTP design against five software approaches (both partitioned, and non-partitioned) over five different datasets. Experiments show a sharp decline in performance for the software approaches as the cardinality increases. The FGL MTP design's throughput is unaffected by the benchmark's cardinality, and can sustain between 500 and 1,500 MTuples/sec depending on the key distribution, achieving an average of 3.3x speedup over all CPU implementations. Due to the limited clock frequency, memory bandwidth,and limited on-chip memory of the FPGA platform relative to state-of-the-art multi-core CPUs, we cannot demonstrate superior raw throughput. However, this proof-of-concept work demonstrates throughput improvements achieved by efficient memory bandwidth utilization using latency-masking threads.

| Cycle | Key | Filter CAM | Lock CAM | HashTable | Comments |
|---|---|---|---|---|---|
| 1 | A | *Miss, Insert (A,1)* {(A,1)} | {} | {} | Request to search key A in HT is sent |
| 2 | C | *Miss, Insert (C,1)* {(A,1),(C,1)} | {} | {} | Request to search key C in HT is sent |
| 3 | A | *Hit, Update A* | *Miss, Thread 1 locks hash(A)* | {} | Key A not found in HT, $Bucket_{hash(A)}$ is locked |
| 4 |  | *Thread 1 clears key A* {(C,1)} | *Hit, hash(A)=hash(C)* {hash(A)} | {(A,2)} | Create new entry (A,2) in HT Key C not found in HT, *Thread 2* waits for lock |
| 5 | B | *Miss, Insert (B,1)* {(C,1),(B,1)} | *Thread 1 frees lock on hash(A)* {} | {(A,2)} | Request to hash(C) in HT is sent *Thread 2 restarts at previous address* |
| 6 | A | *Miss, Insert (A,1)* {(C,1),(B,1),(A,1)} | {} | {(A,2)} | Request to search key B in HT is sent *Thread 2 reaches end of list* |
| 7 |  | {(C,1),(B,1),(A, 1)} | *Thread 2 locks* $node(A).next$ {$node(A).next$} | {(A,2)} | Request to search key A in HT is sent $node(A).next$ is locked |
| 8 |  | *Thread 2 clears key C* {(B,1),(A,1)} | *Thread 2 frees lock for key C* {} | {(A,2),(C,1)} | Create new entry (C,1) in HT Key B not found in HT, CAM busy |
| 9 |  | *Thread 5 clears key A* {(B,1)} | *Thread 4 locks hash(B)* {hash(B)} | {(A,3),(C,1)} | Key A found Update key A in HT to (A,3) |
| 10 |  | *Thread 4 clears key B* {} | *Thread 4 frees lock for key B* {} | {(A,3),(C,1),(B,1)} | Create new entry (B,1) in HT |

Table 3.1: Example showing the contents of the Filter CAM, Lock CAM and HashTable (HT) and *modifications* altering all of them.

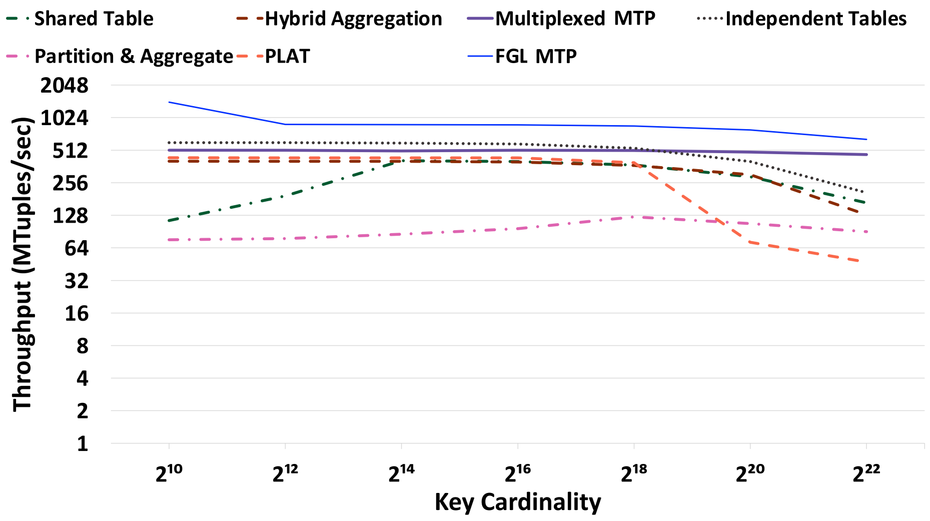


(a) Uniform

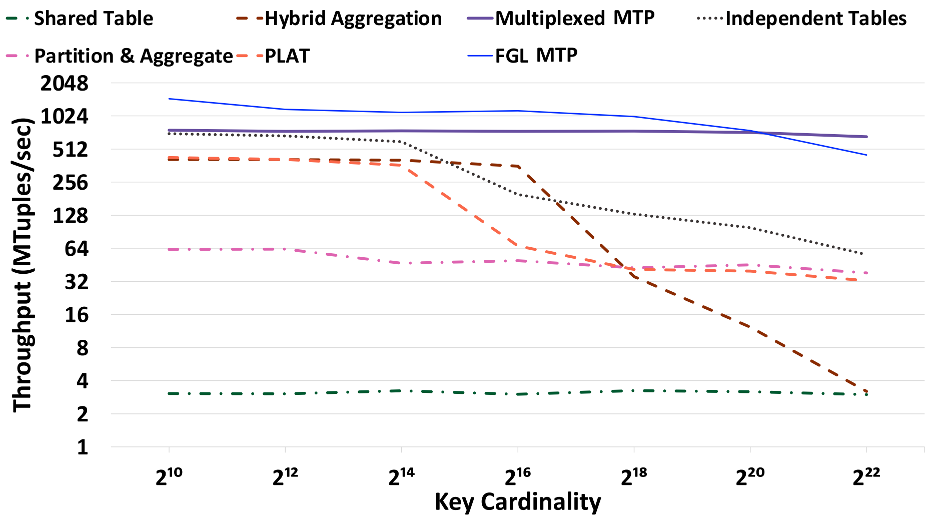


(b) Self Similar

Figure 3.5: Aggregation throughput of hardware and software approaches for datasets with 256M tuples. Y-axes are logarithmic.



(a) Moving Cluster



(b) Heavy Hitter

Figure 3.6: Aggregation throughput of hardware and software approaches for datasets with 256M tuples. Y-axes are logarithmic.

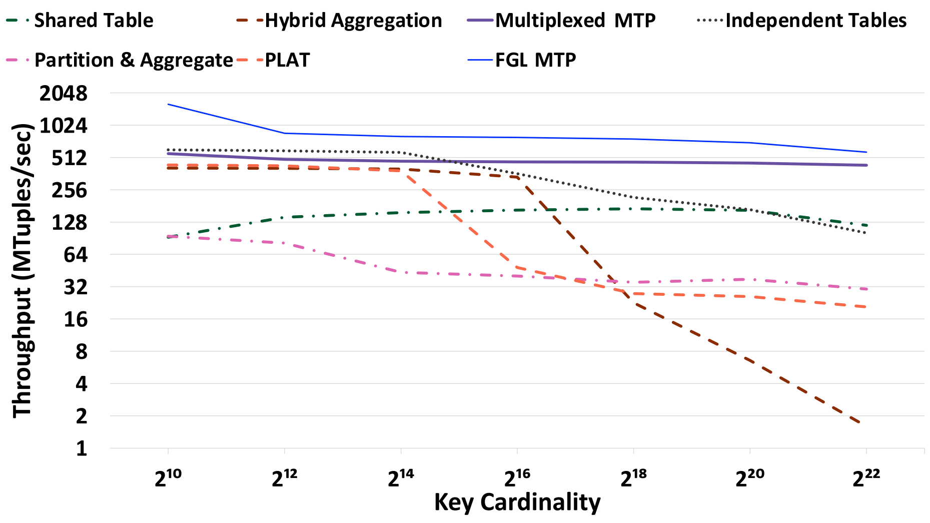


Figure 3.7: Aggregation throughput of hardware and software approaches for Zipf 0.5 dataset with 256M tuples. Y-axes are logarithmic.

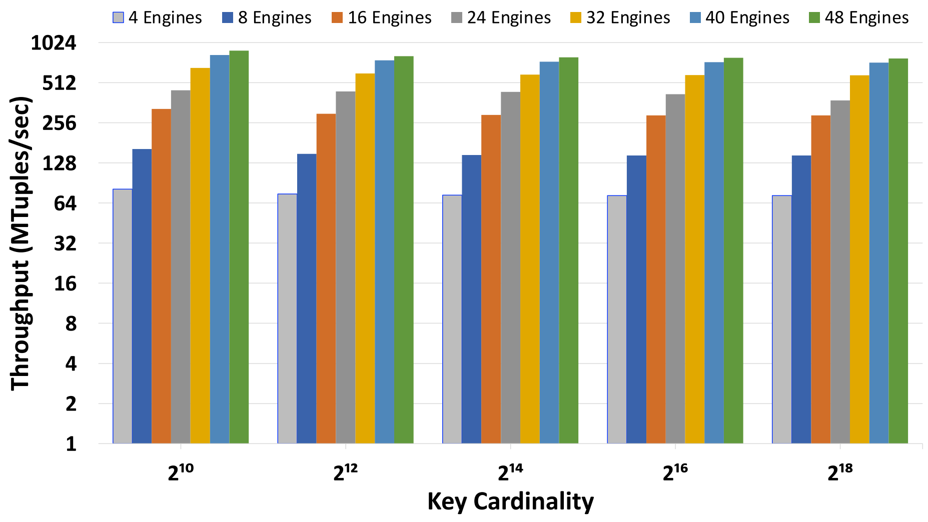


Figure 3.8: The effect of varying the number of engines on throughput from 4 to 48 engines on all 4 FPGAs for Uniform key distribution dataset with 256M tuples. Y-axis is logarithmic.

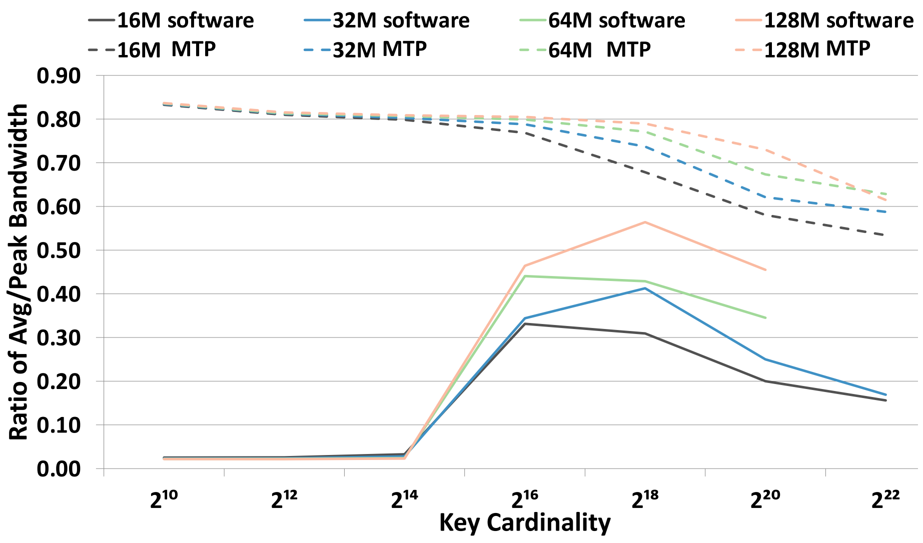
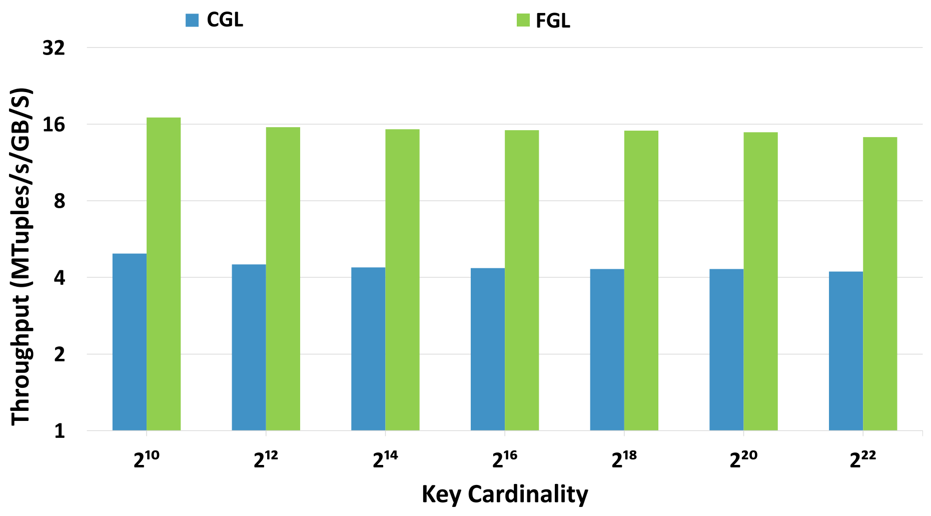


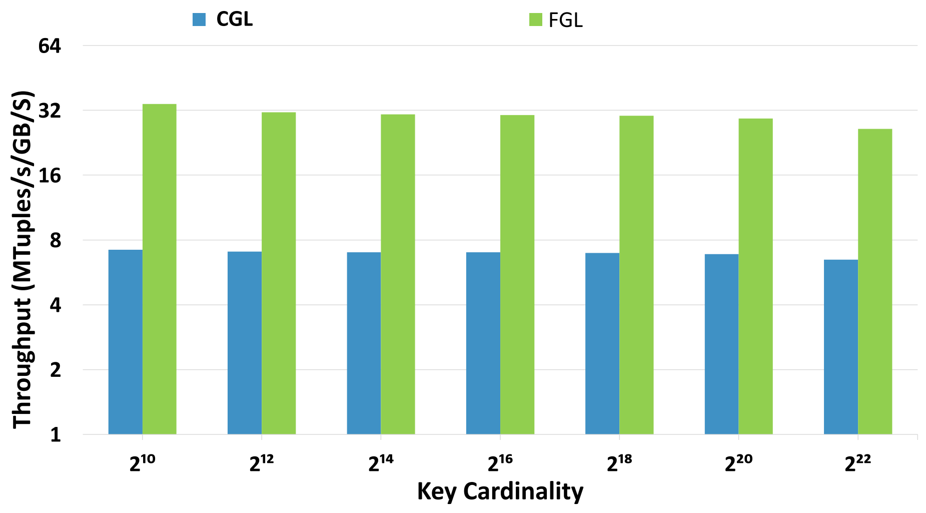
Figure 3.9: Ratio of average effective memory bandwidth to peak theoretical bandwidth achieved by the Independent Tables software algorithm and the FGL design for varying dataset sizes and key cardinalities.

| | # PEs | Registers | LUTs | BRAMs |
|---|---|---|---|---|
| Original | 1 | 99,597 (11%) | 87,194 (18%) | 126 (17%) |
| MUX | 6 | 179,641 (18%) | 200,175 (42%) | 250 (34%) |
| FGL | 12 | 240,118(25%) | 296,778 (62%) | 192 (26%) |

Table 3.2: Per-FPGA resource utilization for aggregation engines.



(a) Uniform



(b) Heavy Hitter

Figure 3.10: Aggregation throughput normalized against available bandwidth of CGL and FGL approaches for Uniform key and Heavy Hitter key distribution datasets with 256M tuples.

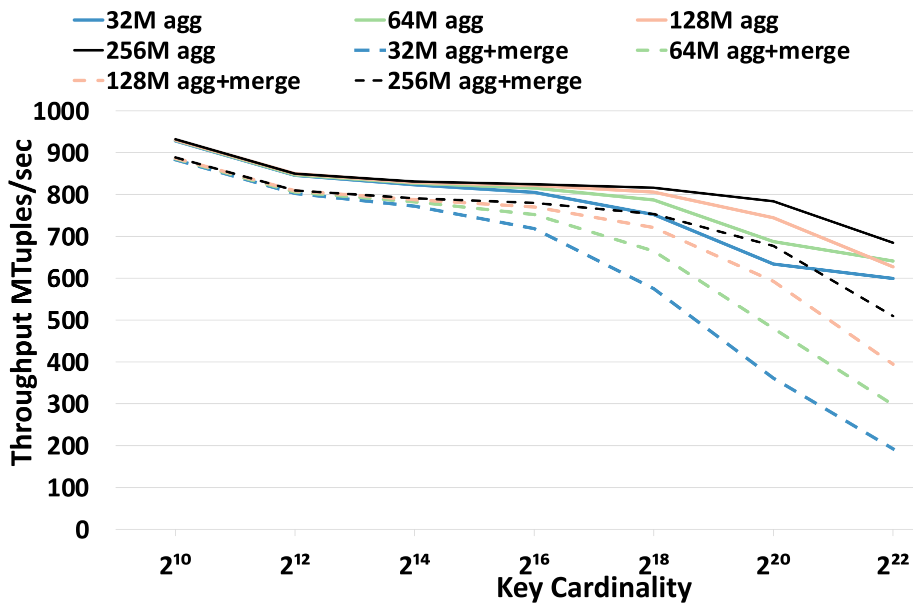


Figure 3.11: Effect of varying relation sizes on the MTP aggregation throughput for datasets with Uniform key distribution. Solid lines represent throughput of the aggregation step (without merge operation), while dashed lines represent end-to-end (aggregation followed by the merge) throughput.

Chapter 4

# High-Performance Parallel Radix Sort on FPGA

## 4.1 Introduction

Sorting large datasets remains one of the classic challenges for high-performance computing. The familiarity of sorting algorithms belies its importance in many crucial applications in the age of big data. Comparison-based and distribution-based sorting algorithms are the two common methods of sorting. Merge sort, an example of the comparison-based approach, has $\mathcal{O}(N \log N)$ time complexity while radix sort, an example of the distribution-based sorting, has $\mathcal{O}(N * B)$ complexity where $B$ is the number of iterations needed to sort a key.

As the technology evolves, new hardware architectures and platforms induce a re-evaluation of sorting algorithms and their implementations on these new architectures. Because the parallel versions of radix sort rely on building and storing a histogram for each Processing Element, it was considered unsuitable for FPGA acceleration as it required too many, costly, reads and writes to memory. However, new FPGA architectures such as the Xilinx UltraScale+ series [10] and the Intel Stratix 10 DX [3] and Agilex I-Series [2] come with very large on-chip storage that makes it worthwhile to re-evaluate radix sort on FPGAs. In this chapter we describe, implement and evaluate Hardware Accelerated Radix Sort (HARS), a parallel implementation of radix sort on an FPGA that takes advantage of this large on-chip storage. Our design enables dividing the unsorted dataset among parallel engines without the need for a merge step. HARS is implemented on Micron's SB-852 FPGA board using Xilinx UltraScale+ XCVU7P FPGA with 64 GB of on-board memory. For in-memory sorting, HARS achieves a throughput of 44 Million 128-bits records per second and is 1.4x faster than a Xeon E5-2640 CPU. Because a GPU, in this case the Nvidia TITAN X Pascal, enjoys a memory bandwidth that is much higher than that on the FPGA board, the raw throughput is about 5.4x that of HARS. However, when normalized to the available memory bandwidth, HARS achieves 1.36x higher throughput.

The main contributions of this chapter are:

- A novel parallel in-memory radix sort implementation on FPGA that does not rely on sorting networks and avoids a final, performance limiting, merge step.
- The size of the sorted data is not restricted by the available on-chip memory.
- A constant throughput that is not dependent on the size of data being sorted and scales with on-chip memory and bandwidth.

## 4.2 FPGA-based Radix Sort Accelerator

Our proposed hardware accelerator is based on a parallel version of the serial Radix Sort algorithm [153]. Our targeted dataset is randomly generated records that consist of key-value pairs. We sort two types of 128-bit records: 1) 80-bit key and 48-bit value (or pointer). 2) 64-bit key and 64-bit value. The Indy category of the Terasort [70] considers records of fixed size of 100 byte records where the keys are 80 bits. Previous work on FPGA-based sort accelerator [71] has considered using a 48-bit pointer to the 90 byte value instead of moving the actual value. This reduces the problem of sorting each 100 byte record into sorting a 16 byte record. The only disadvantage is the number of extra cycles needed to de-reference the pointer to access the data.

### 4.2.1 Parallel Radix Sort Algorithm

Radix sort is based on using Counting Sort as a subroutine. Unlike comparison based sort which has $O(n \log n)$ time complexity, Counting Sort has a linear time complexity. Sorting $N$ $k$-bit keys requires $2N$ storage space and a $2^k$ deep histogram. The time complexity of radix sort is related to: 1) the number of keys to be sorted; 2) the number of digits or bits per key $k$, and; 3) the radix $r$.

In Radix Sort, a set of $N$ keys where each key is composed of $k$-bits is sorted by dividing it into a radix of $r$-bit and sorting the keys based on $r$-bit per iteration until all $k$-bits have been scanned. The Radix parameter $r$ controls how many iterations are needed to sort $k$-bit keys. For instance, sorting 80-bit key is done by dividing it into 16-bit digits, radix 16. In the first iteration keys are sorted based on the first 16-bit, starting from LSB, then they will be sorted based on the next 16-bit and so on until all 80-bit are scanned after five iterations (Equation 4.1).

$$\text{Iterations} = \left\lceil \frac{\text{Total number of bits per key (k)}}{\text{Number of radix bits (r)}} \right\rceil \tag{4.1}$$

Each iteration in our parallel Radix Sort consists of three stages shown in Figure 4.1.

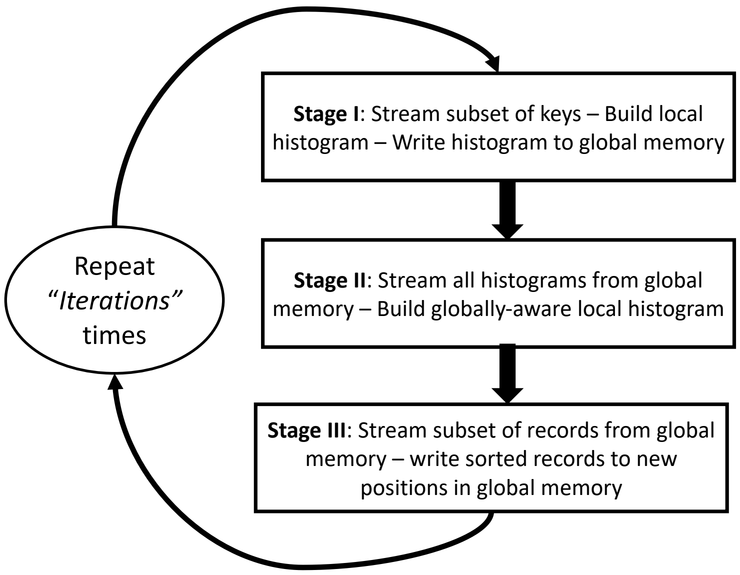

**Stage I**: Stream subset of keys – Build local histogram – Write histogram to global memory
**Stage II**: Stream all histograms from global memory – Build globally-aware local histogram
**Stage III**: Stream subset of records from global memory – write sorted records to new positions in global memory
Repeat *"Iterations"* times

Figure 4.1: The three stages of each iteration in radix sort where the number of iterations is determined by Equation 4.1

The parallel Radix Sort algorithm, executing on $P$ Processing Engines (PEs), is described in Algorithm 1. It consists of three stages:

- Stage I, shown in Algorithm 2, $pSize$ keys (where $pSize = N/P$) are streamed from the global memory to each PE in order to build the histogram table in the local memory. At the end of this stage, all the local histograms are written back to the global memory. The time complexity is $O(N/P)$.

- Stage II, shown in Algorithm 3, each PE reads all the locally generated histograms to its own local memory performing a prefix sum on all the histograms. Each PE re-streams the histograms of subsequent PEs and adds them as an offset to its histogram. The time complexity is $O(P2^r)$. For $r \ll N$ the time complexity of stage II is negligible.

- Stage III, shown in Algorithm 4, each PE streams again its subset of both keys and values. The radix bits of each key are compared against the updated local histogram per PE and the new write location is calculated. The write address of key-value pair can be anywhere in the global memory. It is not restricted to the range of data subset assigned to the PE. The Time complexity is $O(N/P)$.

**Example:** Let the set KEYS be an unsorted dataset, $N = 8$, $k = 4$, and $r = 2$. Sorting this set requires two iterations. Two PEs that share the same memory space will be used to demonstrate the algorithm. Each PE also has its own local memory.

$$\text{KEYS} = \{0110, 1101, 1010, 1110, 1010, 0111, 0110, 1100\}$$

In iteration 1, the first two least significant bits of each key are scanned to construct a local histogram in each PE. The depth of the histogram table per PE is $2^r = 4$.

**Stage I:** this stage begins with each PE streaming its subset of $pSize = 4$ keys. PE 1 streams:

$$\text{KEYS}[0 \rightarrow 3] = \{0110, 1101, 1010, 1110\}$$

While PE 2 streams:

$$\text{KEYS}[4 \rightarrow 7] = \{1010, 0111, 0110, 1100\}$$

Table 4.1 shows the final local histogram of PEs 1 & 2. All PEs write their local histograms to the global memory at the end of the first stage.

| Pattern | Count PE 1 | Count PE 2 |
|---|---|---|
| 00 | 0 | 1 |
| 01 | 1 | 0 |
| 10 | 3 | 2 |
| 11 | 0 | 1 |

Table 4.1: Local Histogram for PEs 1 & 2

**Stage II:** all local histograms have to be exchanged between the PEs. Each PE builds a local histogram that is aware of the other PEs' histograms. This ensures that keys are sorted in global scope without any conflict in ordering. Hence, eliminating the need for merging the sorted $P$ subsets at the end. At the end of stage II, each PE will have completed building a local globally-aware table.

| Pattern | Count PE 1 | Count PE 2 |
|---|---|---|
| 00 | 0 | 0 |
| 01 | 1 | 2 |
| 10 | 2 | 5 |
| 11 | 7 | 7 |

Table 4.2: Updated Local Histogram, PE 1 & PE 2

**Stage III:** both keys and values are streamed as shown in Algorithm 4. The first two bits per each key are compared against the corresponding entry in the local histograms shown in Table 4.2 in PE 1 and PE 2. For example in PE1, when $key[0] = 0110$ is streamed, the entry in Table 4.2 for bits 10 is 2. So, $key[0] = 0110$ will be written to global memory at location 2 and the entry for bits 10 is incremented to 3. $key[1]$ will be written to location 1 and $key[2]$ to location 3 in global memory. When $key[3] = 1110$ is streamed, the entry table for bits 10 is now 4. So, $key[3]$ will be written to location 4 in the global memory. The same process happens simultaneously in PE2.

After all PEs have completed their writes, the second iteration begins. Each PE steams the same range and size of keys from the memory location where previous writes took place. PE 1 streams:

$$\text{KEYS}[0 \rightarrow 3] = \{1100, 1101, 0110, 1010\}$$

while PE 2 streams:

$$\text{KEYS}[4 \rightarrow 7] = \{1110, 1010, 0110, 0111\}$$

The same three stages are executed in a second iteration to sort the records based on next and final two bits. At the end of the second iteration the set KEYS is fully sorted.

$$\text{KEYS} = \{0110, 0110, 0111, 1010, 1010, 1100, 1101, 1110\}$$

**Algorithm 1:** Parallel Radix Sort

```
Input: Unsorted N pairs of Key-Value
Result: Sorted N pairs of Key-Value
Definitions
    $P$ ←Number of PEs
    $N$ ←Number of records of Key-Value pairs
    $K$ ←Number of bits per key
    $r$ ←The Radix (scanned bits)
    PE's size: $pSize \leftarrow N/P$
    PE's ID: $ID_{PE}$
    Number of iterations: $Itration \leftarrow \lceil K/r \rceil$
    Key: $N$ length {unsorted, partially sorted, or sorted}.
    Key-Value:$N$ length {unsorted, partially sorted, or sorted}.
    Bin: $2^{16}$ Local histogram vector per PE
    Table: $P \times 2^r$ array stored in global memory
end
Initialization:
    for $i \leftarrow 0$ to $2^r - 1$ by $1$ do
        $Bin[i] \leftarrow 0$
    end
end
for $k \leftarrow 0$ to $Itrations - 1$ by $1$ do
    I: BUILD LOCAL HISTOGRAMS:
    Each PE executes in parallel:
        Stage I: Refer to Algorithm. 2.
    end
    II: BUILD LOCAL GLOBALLY-AWARE HISTOGRAMS:
    Each PE executes in parallel:
        Stage II: Refer to Algorithm. 3.
    end
    III: COMPARE & WRITE:
    Each PE executes in parallel:
        Stage III: Refer to Algorithm. 4.
    end
end
```

### 4.2.2 FPGA Platforms

Two platforms were used in evaluating the proposed hardware accelerator HARS: the Micron WX-2000, hereafter referred to Wolverine I (or W1), and the SB-852 here called Wolverine II (or W2).

In both boards, the virtual address space is shared between the host and the co-processor, through Globally Shared Virtual Memory, and appears to the x86 host's memory system as another processor connected over PCIe bus. Table 4.3 highlights the main differences between two boards.

```
Algorithm 2: I: Build Local Histograms
Input: Key{unsorted or partially sorted}, $pSize, ID_{PE}, k$
Output: $Table$
Each PE executes in parallel:
    for $i \leftarrow 0$ to $pSize - 1$ by $1$ do
        $radix \leftarrow Key[i] << (r \times k)$
        $Bin[radix] \leftarrow Bin[radix] + 1$
    end
    for $i \leftarrow 0$ to $2^r - 1$ by $1$ do
        $Table[ID_{PE}][i] \leftarrow Bin[i]$
        $Bin[i] \leftarrow 0$
    end
end
```

```
Algorithm 3: II: Build Local Globally-Aware Histograms
Input: $Table, P, ID_{PE}$
Each PE executes in parallel:
    for $j \leftarrow 0$ to $P - 1$ by $1$ do
        for $i \leftarrow 0$ to $2^r - 1$ by $1$ do
            $Bin[i] \leftarrow Bin[i] + Table[j][i]$
        end
    end
    for $i \leftarrow 1$ to $2^r - 1$ by $1$ do
        $Bin[i] \leftarrow Bin[i] + Bin[i-1]$
    end
    for $i \leftarrow 2^r - 1$ to $1$ by $-1$ do
        $Bin[i] \leftarrow Bin[i-1]$
    end
    $Bin[0] \leftarrow 0$
    for $j \leftarrow 0$ to $ID_{PE} - 1$ by $1$ do
        for $i \leftarrow 0$ to $2^r - 1$ by $1$ do
            $Bin[i] \leftarrow Bin[i] + Table[j][i]$
        end
    end
end
```

The larger overall on-chip memory available on W2 permits using higher radix for the sort and hence reduces the number of required iterations. The UltraRAM on the W2's Xilinx Ultra-Scale+ [9] is used for this purpose. W1 and W2 have the same architecture. Figure 4.2 shows the architecture of W2. The architecture of W1 is identical but using Xilinx Virtex 7 chip with DDR3 memory instead.

In both platforms the co-processor, a single Xilinx FPGA chip, is connected to its global on-board memory through a distributed crossbar; four memory controllers manage 32 half-duplex Memory Channels (MCs), as shown in Figure 4.3.

**Algorithm 4:** III: Compare & Write

```
Input: Key-Value {unsorted or partially sorted}, $pSize, k$
Output: Key-Value {partially sorted or sorted}
1  Each PE executes in parallel:
2      for $i \leftarrow 0$ to $pSize - 1$ by 1 do
3          $radix \leftarrow Key[i] << (r \times k)$
4          $idx \leftarrow Bin[radix] + 1$
5          $Bin[radix] \leftarrow Bin[radix] + 1$
6          $\{Key[idx], Value[idx]\} \leftarrow \{Key[i], Value[i]\}$
7      end
8      for $i \leftarrow 0$ to $2^r - 1$ by 1 do
9          $Bin[i] \leftarrow 0$
10     end
11 end
```

| | | W1 (XC7V2000T) | W2 (XCVU7P) |
|---|---|---|---|
| **On-chip Memory (Mb)** | **Total** | 45.4 | 230.6 |
| | **BRAM** | 45.4 | 50.6 |
| | **URAM** | N/A | 180 |
| **Global Memory (GB)** | | 32 DDR3 | 64 DDR4 |
| **Clock Freq. (MHz)** | | 168.7 | 266.67 |
| **Bandwidth (GB/s)** | **Peak** | 42 | 64 |
| | **Measured** | 32 | 42 |

Table 4.3: Characteristics of Micron's W1 and W2 boards

In both W1 & W2, the host server has two Intel Xeon E5-2640 processors running at 2.60 GHz with 20 MB L3 cache and 128 GB of DDR4 RAM.

## 4.2.3 HARS' Architecture

Our proposed accelerator HARS is a set of PEs implemented on the FPGA as illustrated in Figure 4.3. Each PE is capable of executing all three stages of the Radix Sort algorithm.

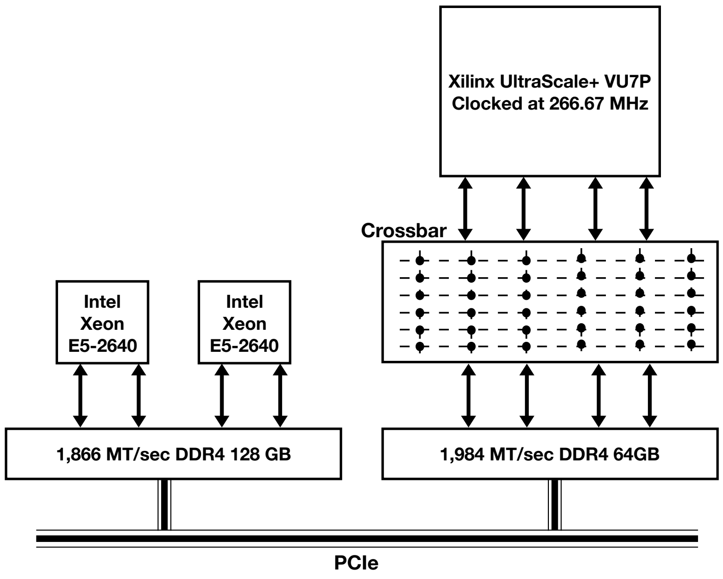


Figure 4.2: Host and accelerator in Micron's Wolverine II (W2) platform

The number of possible PEs is determined by the number of available MCs on the FPGA. In order to produce an output every cycle, 2 MCs per PE are needed hence 16 PEs can be synthesized on the FPGA. All local histograms are implemented using on-chip memory shown in Table 4.3.

Figures 4.4, 4.5, and 4.6 show the implementations of the three stages that correspond to the Algorithms 2, 3, and 4 respectively.

The host server orchestrates the sequencing of the three stages as shown in Figure 4.1. The host application acts as threads barrier ensuring that all PEs have completed executing the same stage before initiating the next one.

**Stage I Circuit (Building Local Histograms)** Data flow for this circuit is shown in Figure 4.4. Each PE has a local histogram of depth $2^r$ and width of 32-bit. W1 platform is limited to a histograms of size $14 \times 2^{14}$ bits per PE. In W1, the histograms are built entirely using BRAM. W2

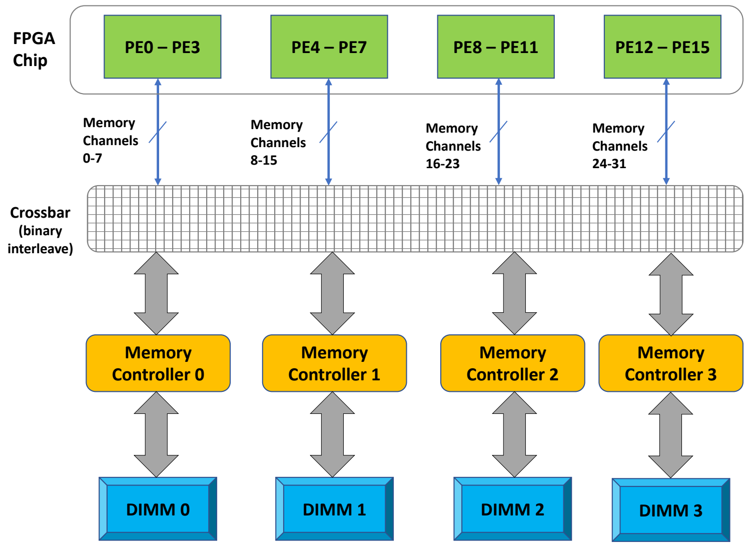


Figure 4.3: Accelerator's system architecture

platform has local histograms with $16 \times 2^{16}$ bits per PE built using a mix of BRAM and URAM blocks. In both designs, about a third of the BRAM resources are used in building the accelerator's memory and PCIe interfaces.

Upon receiving the starting signal, read requests for keys are issued continuously as long as there is space in the input FIFO and there is no memory stall request. Keys are continuously popped from the input FIFO and shifted based on $r$ to extract the pattern scanned for that specific iteration from the key. The pipelined design ensures that while a key is being read from the input FIFO, a bin is being read from memory based on a previous key while another bin in memory is being incremented. Read after write data hazard is handled in the design. Once all the keys have been processed in a PE, the local histogram is written to the global memory.

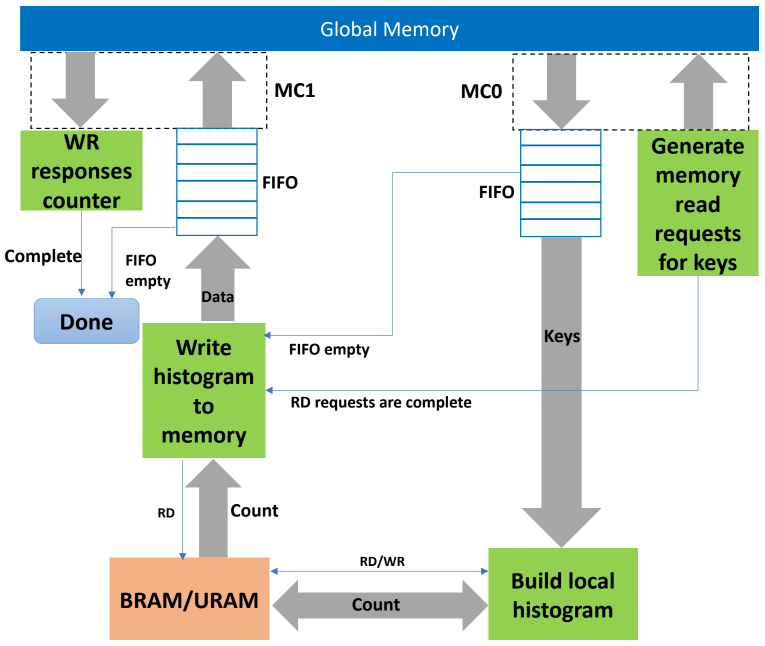


Figure 4.4: Data-flow for Stage I per PE: building and writing the local histogram

**Stage II Circuit (Building Local Globally-Aware Histograms)** The data flow for this circuit is shown in Figure 4.5. Each PE rebuilds a new histogram such that the new histogram contains the offsets needed to sort the assigned subset of keys in accordance with their order in the global dataset.

**Stage III Circuit (Compare & Write)** The data flow for this circuit is shown in Figure 4.6. Each PE generates stream requests for all key-value pairs in its $pSize$ subset, calculates the new address of the key-value pair, and issues the write request.

### 4.2.4 Resources Utilization

Table 4.4 shows the resource utilization (registers, LUTs,and memories used) for HARS on the W2 platform. HARS utilizes most available on-chip Block and Ultra memory for a radix

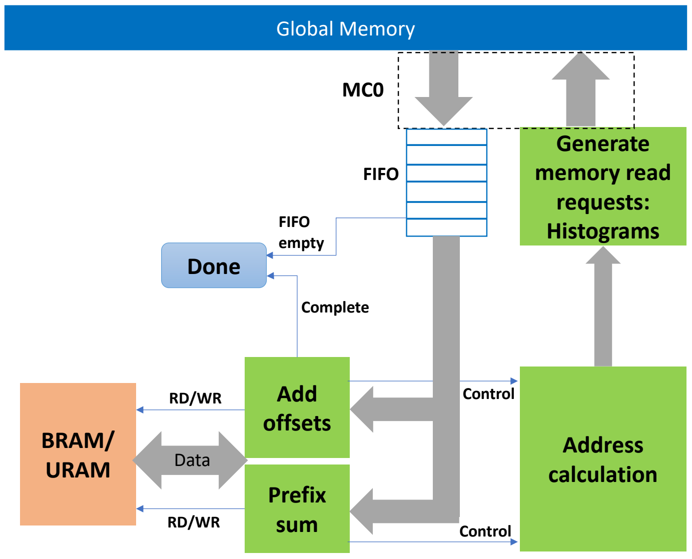


Figure 4.5: Data-flow for Stage II per PE: synchronizing and updating histograms

value of $r = 16$. The next radix value that reduces the number of iterations is $r = 20$ which exceeds the total memory resources available on-chip.

## 4.3 Performance Evaluation

HARS was implemented on both the W1 and W2 FPGA boards. Since HARS is designed as an in-memory sorting application, all measurements were done after the data was loaded to global memory.

### 4.3.1 Sorting on FPGA and CPU

The CPU software evaluation was performed on the host server described in 4.2.2 using single CPU chip with two configurations: 8 and 4 cores. In our experiments, the frequency

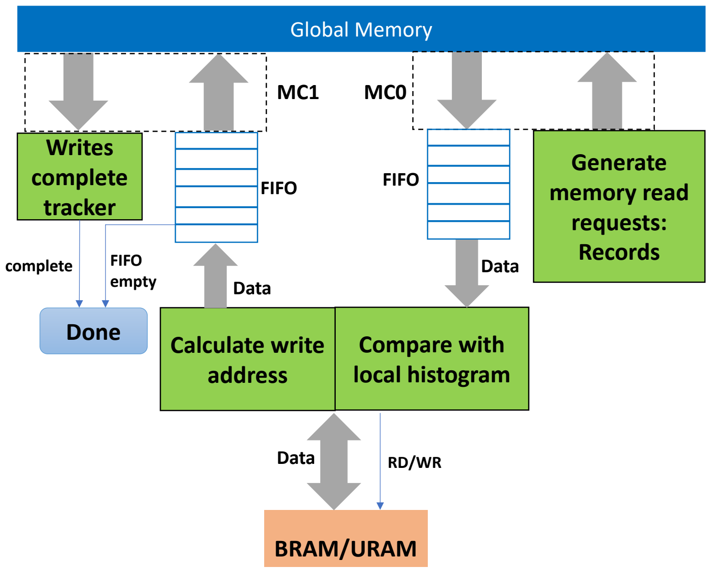


Figure 4.6: Data-flow for Stage III per PE: compare & write

scaling is switched off and the maximum frequency of 2.6 GHz is used. For the CPU implementation, we use *__gnu_parallel::sort* which is a widely-used parallel software implementation of C++ STL's std::sort() based on libstdc++. The C++ STL sort routine has an average time complexity of $\mathcal{O}(N \log N)$ [8]. GNU G++ 4.8.5 with C++11 standard is used to compile the host application with OpenMP and optimization flags *-fopenmp* and *-O3*. The host server runs CentOS 7.6.

The average throughput for various sizes of data sets is examined in order to determine the effect on throughput across FPGA and CPU implementations. When considering data sets of 128-bit records with 80-bit keys, Figures 4.7 and 4.8 show that the average throughput increases with the size of data sets for both CPU and FPGA implementations respectively. For the CPU, the throughput peaks around 536M records at 33.5M rec/sec for the 8 cores configuration. The throughput starts to decline slowly after 536M records. The same trend is observed with 4 cores. Doubling the number of CPU cores from 4 to 8 results in 1.6x higher throughput on average. Sorting

| **Resource** | **Available Blocks** | **(%) Utilization Total** | **(%) Utilization Per PE** |
|---|---|---|---|
| **Registers** | 1,576K | 25.79 | 1.6 |
| **Lookup Tables (LUT)** | 788K | 34.04 | 2.1 |
| **LUT RAMs** | 395K | 13.51 | 0.8 |
| **Block RAMs** | 1,440 | 75 | 4.7 |
| **Ultra RAMs** | 640 | 20 | 1.3 |
| **Memory Channels** | 32 | 100 | 2 |

Table 4.4: Post P&R Resources Utilization

throughput for FPGA is stable at 43M rec/sec and increases linearly as the data set size increases. No tests beyond 2B records were possible on the FPGA due to the size of available memory on W2 board.

To evaluate the effect of key size on sorting time we experimented sorting datasets with key sizes 64 bits and 80 bits as shown in Figure 4.9.

Both HARS and the CPU show better performance for smaller key sizes. On the CPU, sorting 64-bit keys is 1.3x faster than 80-bit keys, while on HARS it is 1.25x faster. A smaller key size for the same radix implies fewer iterations in radix sort (Equation. 4.1).

The differences between the W1 & W2 platforms are shown in Table 4.3. Table 4.5 shows the variation in HARS performance on W1 & W2. The 1.5x gain in performance is due to three

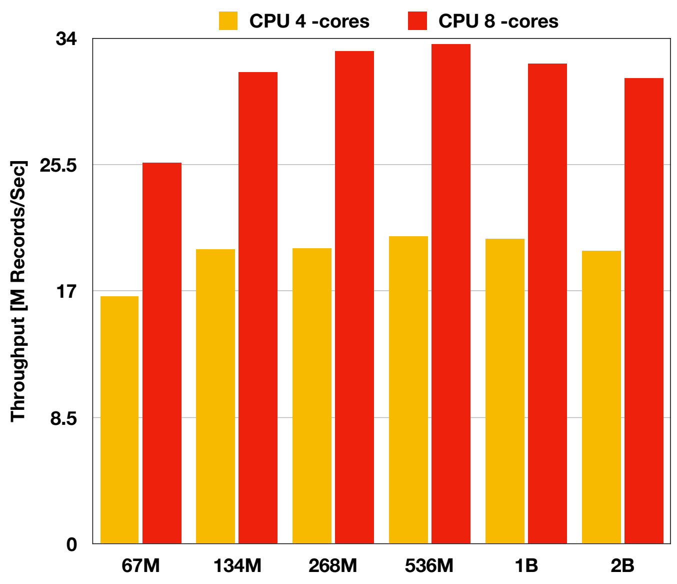


Figure 4.7: Average sorting throughput on both 8 and 4 cores CPU

factors: (1) larger on-chip memory enables the use of higher radices and hence fewer iterations and shorter sort time; (2) higher clock frequency; and (3) higher memory bandwidth.

Figure 4.10 shows the stable throughput of HARS when implemented on W2 compared with two CPU configuration. HARS is 1.4x and 2.2x faster than CPU with 8 & 4 cores respectively.

| **Platform** | **Radix** | **Throughput (million records/sec)** |
|---|---:|---:|
| **W1** | 14 | 29 |
| **W2** | 16 | 44 |

Table 4.5: Throughput comparison between HARS on W1 & W2 FPGA platforms

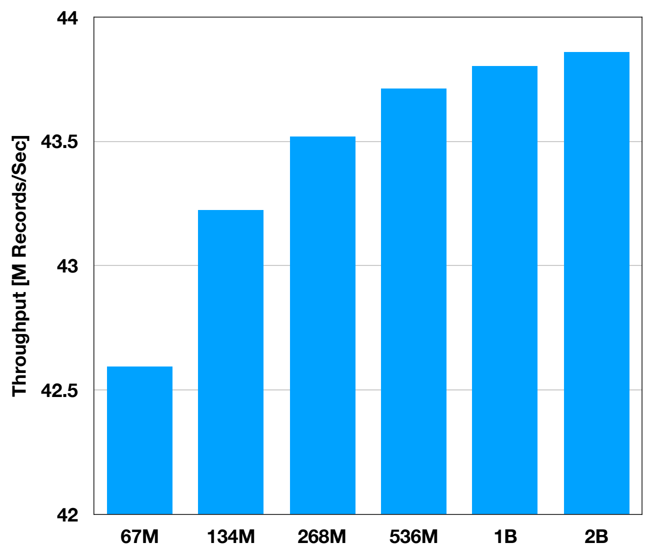


Figure 4.8: Average sorting throughput using HARS on W2 platform

### 4.3.2 Sorting on GPU

GPU experiments were performed on Nvidia TITAN X Pascal, running at 1,417 MHz [5].

On GPUs, there are two widely used sorting routines from Thrust library: (1) *Thrust::sort* which sorts keys only, and (2) *Thrust::sort_by_key* which sorts key-value pairs.

*Thrust::sort* can be used to sort key-value pairs by defining keys using CUDA built-in vector type *ulong2* that is 128-bit which stores both key and value. Sorting is done using mask along with a user-defined compare function that is passed to *Thrust::sort*. Our experiments indicate that using *thrust::sort* in the aforementioned configuration yields throughput that is 1.7x higher than that when *Thrust::sort_by_key* is used so we use *Thrust::sort* to produce the GPU throughput results. When using *ulong2* datatype and user-defined compare function in *thrust::sort*, thrust library switches from radix sort to merge sort. The GPU software was compiled with NVCC version 10.0.130.

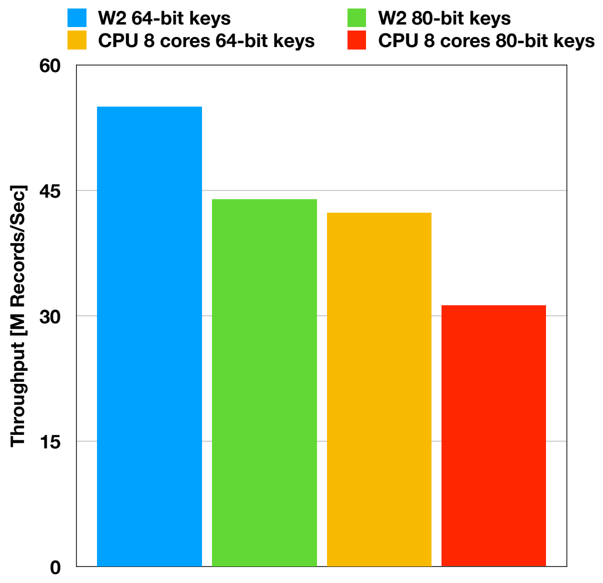


Figure 4.9: Sorting throughput in records/sec for both 80-bit and 64-bit keys using HARS and CPU (2B records)

Table 4.6 shows a comparison in capabilities between W2, CPU 8-cores, and GPU platforms. The overhead of moving data between the host and GPU main memory is not considered when evaluating GPU performance. In terms of throughput per second, Table 4.7 shows that the GPU has the highest throughput comparing to both W2 and CPU. Since each of the platforms used in the comparison runs on a different clock frequency, examining throughput in terms of records/cycle illustrates which platform has higher efficiency per cycle. Figure 4.11 shows the sorting throughput on the same dataset normalized to clock frequency for GPU, W2 and CPU. GPU & HARS nearly have the same throughput despite the GPU having 7.5X more bandwidth.

Since GPU has much higher memory bandwidth, the throughput in Figure 4.12 is normalized by bandwidth in order to measure the efficiency of each platform. Even GPU throughput per second is 5.4X higher than that of W2, when throughput is normalized by bandwidth, W2 is 1.2X

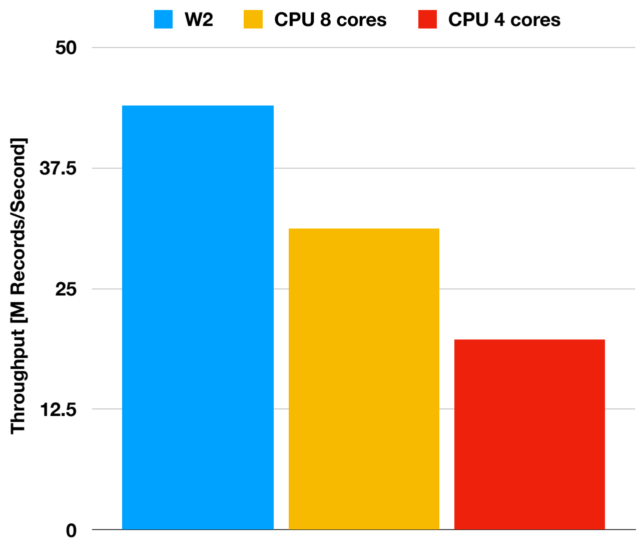


Figure 4.10: Throughput of HARS and 8 & 4 cores CPU (2B records)

and 1.36X higher than CPU and GPU respectively. HARS uses both bandwidth and clock cycles more efficiently than both CPUs and GPUs

### 4.3.3 Lessons Learned

The availability of on-chip memory makes it possible to store the histograms locally and reduce the traffic to/from on-board global memory. Using higher radices ($r$) reduces the number of required iterations in the radix sort algorithm, but increases the sizes of the local histograms. Having more PEs increases the parallelism but requires more area on the FPGA(s).

Our experimental evaluation has shown that:

- HARS delivers a constant throughput irrespective of the dataset size, unlike the CPU (Figures 4.7 & 4.8). HARS is therefore more scalable than the CPU implementation.

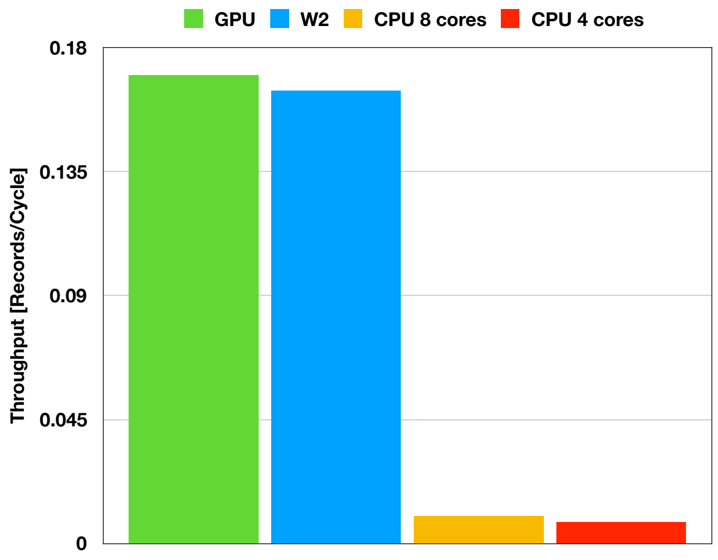


Figure 4.11: Throughput in terms of records/cycle for HARS, GPU and 8 & 4 cores CPU (268M records)

- The HARS throughput, measured in records/cycle, is comparable to that of the GPU and 16.4x times larger than that of the CPU hence demonstrating the computational efficiency of the HARS approach (Figure 4.11).

- Similarly, when throughput is normalized by bandwidth, HARS is more bandwidth efficient than either the CPU or GPU implementations (Figure 4.12) demonstrating HARS's effectiveness in using the available bandwidth.

The Xilinx UltraScale+ XCVU7P used in our evaluation has 230.6 Mb on-chip memory (combined BRAM & URAM). The 345.9 Mb on-chip memory available on the Xilinx UltraScale+ XCVU13P would enable using $r = 20$ radix which would increase the throughput by 1.25x over that of W2.

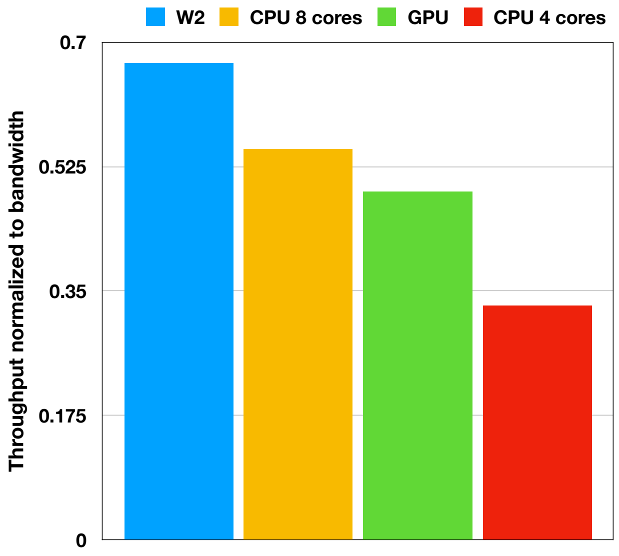


Figure 4.12: Throughput normalized by bandwidth for HARS, GPU and 8 & 4 cores CPU (268M records)

## 4.4 Conclusion

In this chapter we have introduced HARS (Hardware Accelerated Radix Sort) a novel parallel implementation of radix sort on FPGAs that leverages the on-chip memory to locally store the necessary histograms thereby drastically reducing the traffic to/from on-board memory and hence improving the throughput. For in-memory sorting, HARS achieves a throughput of 44 Million 128-bits records per second and is 1.4x faster than the CPU (Xeon E5-2640). Because a GPU, in this case the Nvidia TITAN X Pascal, enjoys a memory bandwidth that is much higher than that on the FPGA board, the raw throughput is about twice that of HARS. However, when normalized to the available memory bandwidth, HARS achieves 1.36x higher throughput.

|  | **Peak Bandwidth (GB/sec)** | **Global Memory (GB)** | **Freq. (MHz)** | **Execution Units** |
|---|---:|---:|---:|---:|
| **GPU TITAN X Pascal** | 480.4 | 12 | 1,417 | 3,584 |
| **CPU E5-2640 v3** | 59 | 128 | 2,600 | 8 |
| **W2** | 64 | 64 | 267 | 16 |

Table 4.6: Comparison of Platforms

|  | **GPU TITAN X Pascal** | **CPU 8 cores** | **W2** |
|---|---:|---:|---:|
| **million records/sec** | 235.5 | 33 | 43.5 |
| **records/cycle** | 0.17 | 0.01 | 0.16 |

Table 4.7: Sorting results for dataset of 268M records for GPU, CPU and W2

# Chapter 5

# HLS Sample Sort Implementation on FPGA

## 5.1 Introduction

Sorting accelerators are indispensable component in hardware database processing systems such as [113, 138]. Various optimizations of sorting software implementations for CPUs [38, 68, 88] and GPUs [15, 55, 43] have been proposed. Several hardware sorting accelerators have been proposed, some utilize sorting networks [99, 130, 92, 34], while others target comparison-based sorting [103, 115, 91] as well as a hybrid of sorting networks and comparison-based sort [71, 122]. Distribution-based sorting on FPGA such as radix sort has also been explored [22].

A common challenge with hardware sorting accelerators is the difficulty in scaling the accelerators for very large datasets. This is due to the limited memory available on chip in embedded system and re-configurable platforms. Traditional methods of sorting a large dataset is to divide the set into smaller sub-arrays such that a sub-array fits in cache memory. Sorted sub-arrays are then merged into larger sub-arrays until the final complete sorted array is produced. The size of merged sub-arrays grow with each merge step until the data cannot fit in cache memory.

The regularity of data access is another challenge in sorting algorithms. In HARS, presented in Chapter 4, the distribution of the keys to the buckets, at each radix, is very irregular, i.e does not exhibit any spatial locality. The FPGA platforms used to implement HARS, Micron's W1 & the W2, both provided a reasonably high throughput for irregular memory accesses because the full crossbar network connecting the FPGA to the global memory. The Xilinx Alveo U280 Data Center accelerator card is designed to support large burst memory accesses [146, 144]. Its memory bandwidth degrades by 92.5% for memory accesses with very large strides [132]. A further evaluation of irregular memory accesses was done by building a histogram using 4-Byte bins. The maximum achievable bandwidth was 13.27 GB/s, a reduction of 95.5% of the maximum achievable bandwidth with regular memory accesses.

This chapter explores a parallel FPGA implementation of sample sort [25], a cache-oblivious sorting algorithm that enables sorting very large datasets using local memory. Sample sort enables all steps to be performed on data small enough to fit in cache memory. It avoids producing large sub-arrays that can not be merged in local memory. In addition, sample sort exhibits only serial memory accesses which makes it suitable for acceleration on the Xilinx Alveo U280 FPGA board. In sample sort, a set of pivots that represents the dataset is utilized to distribute keys from sorted sub-arrays into buckets that can fit on local memory. Finally, sorting the resulting buckets produces the completely sorted set. The new Xilinx FPGA programming environment Vitis supports High-Level Synthesis (HLS) [147, 148] and provides a library of pre-implemented sorting

APIs [139, 140, 141, 142]. Sample sort is implemented entirely using Vitis HLS and deployed on Xilinx Alveo U280 Data Center accelerator card.

The contributions of this chapter can be summarized as follows:

- A novel parallel implementation of sample sort on FPGA.
- The exclusive use of Vitis HLS to implement the sample sort.
- An exploration of the capabilities of Vitis HLS and the Xilinx Alveo board to support this large none-trivial application.

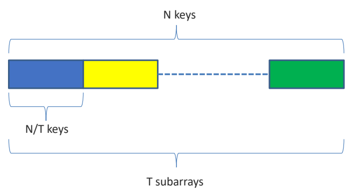
## 5.2 Background

### 5.2.1 Sample Sort

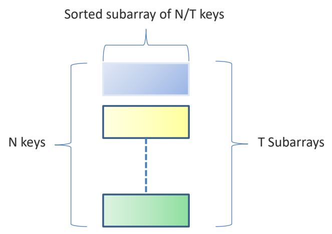
Sample sort is a cache-oblivious sorting algorithm [53, 108] its parallel version is described in [25]. The key advantage of sample sort is enabling sorting very large sets by dividing the work to operate on small sub-sets of data that fit entirely on local memory, e.g. caches in CPUs/GPUs or on-chip memory in FPGAs and ASICs. In this section we will refer to any type of local memory as cache memory.

The main steps of sample sort (Figure 5.1) are summarized as follows:

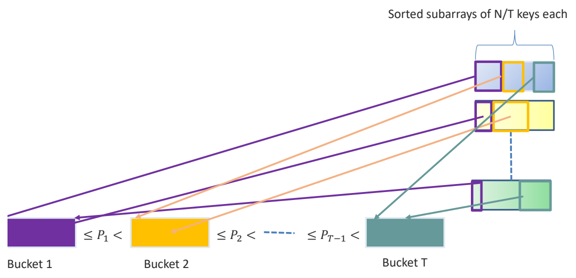
1. Divide unsorted data-set located on off-chip memory into a set of sub-arrays where a sub-array or more fit in cache (Figure 5.1a).
2. Sort each sub-array in locally using a sorting sub-routine (Figure 5.1b).
3. Generate and sort a set of pivots represent that the data.



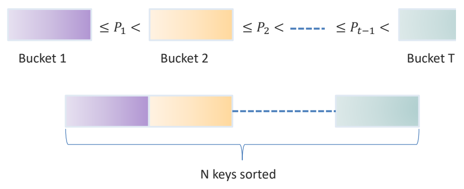
(a) Divide the set of $N$ keys into equally sized $T$ sub-arrays



(b) Sort all $T$ sub-arrays & build the histogram matrix



(c) Distribute keys from each sub-array into their corresponding buckets using prefix-sum



(d) Sort buckets & concatenate them according to the pivots order

Figure 5.1: The main steps of sample sort

4. Use the pivots to build a histogram for each sub-array.
5. Transpose the histogram matrix (divide-and-conquer matrix-transpose algorithm [54]).
6. Prefix-sum is computed from the transposed histogram matrix.
7. Keys are distributed from each sub-array to their corresponding buckets using the prefix-sum (Figure 5.1c).
8. Sort the buckets using a sorting sub-routine.

9. Sorted buckets are concatenated according to the pivots order (Figure 5.1d).

Pivots selection is a key component in sample sort. Deterministic and randomized methods of selecting pivots are proposed in [25]. The optimal pivots set would partition the keys into equally sized buckets. Another criterion for the pivots selection is that the resulting buckets should fit in cache or local memory. In a deterministic selection of pivots, the set is split into $\sqrt{n}$ sub-arrays, each has $\sqrt{n}$ keys. Each $(\log n)$-th key from each sorted sub-array is selected for the samples set. After sorting the sample set, pivots are chosen by selecting each $\sqrt{n}$-th key from the set.

In this chapter we have focused on deterministic pivot selection. Because sample sort does not incur random memory accesses and is highly parallelizable, it is suitable for FPGA implementation.

### 5.2.2 Xilinx Alveo U280 Data Center Accelerator Card

The Xilinx Alveo U280 Data Center accelerator card is one of the new generation of FPGA cards with PCIe interface to the host. The Alveo U280 features an FPGA chip (Figure 5.2) that is built using Stacked Silicon Interconnect (SSI) technology. It consists of three stacked silicon dies, each die is called a Super Logic Region (SLR). SLRs are connected using Super Long Lines (SLL) routing resources [147]. The FPGA chip is divided into two regions: 1) A fixed static region which contains the board's interface and is configured once during the setup of the board. 2 ) A dynamic re-configurable region where the user's logic and memory interfaces are implemented [144]. The default clock frequency for FPGA kernels is 300 MHz, while the maximum supported clock frequency is 450 MHz [146].

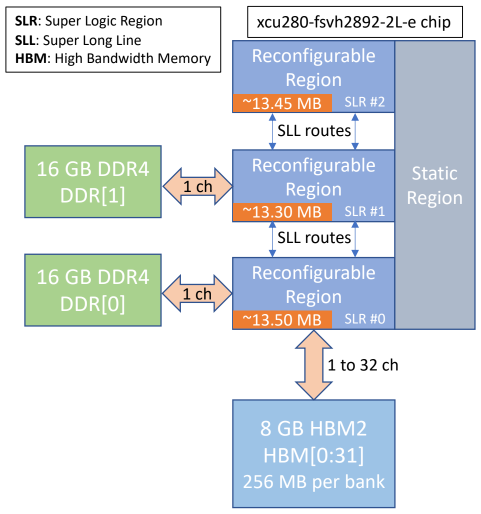


Figure 5.2: Floorplan of U280 FPGA chip. The chip consists of three stacked silicon dies called SLRs. The chip consists of two regions: 1) A static region that contains the board's interface and needs to be configured once. 2 ) A dynamic re-configurable region where the user's kernels and memory interfaces are implemented [144]. SLR0 connects to one bank of DDR4 and all 32 banks of HBM2. SLR1 connects to one bank of DDR4, while SLR2 is not connected to any off-chip memory resources. SLRs are connected by SLL routes. Total on-chip memory (BRAM & URAM) shown on each SLR is extrapolated from [146].

The Alveo U280 card features two types of global off-chip memory: 1) Two banks of DDR4, each bank is 16 GB. 2) Two stacks of HBM2, each bank is 4 GB. Each HBM2 stack is divided into 16 banks, each bank is 256 MB. Each HBM2 bank is interfaced with the FPGA using a Pseudo Channel (PC). Global connection between any AXI port to any HBM bank is made possible by a partial crossbar (switch). A full mini-switch connects each group of four AXI channels with four PCs. Thus, each group of four HBM2 banks can be accessed using a single AXI memory channel without loss in bandwidth. [145, 135, 132]. On the other hand, the bandwidth between an AXI channel and a bank outside the mini-switch it's connected to is significantly reduced.

Each of the three SLRs has a subset of the global off-chip memory interfaces. SLR0 has the interface to all 32 HBM2 banks as well as one interface to bank #0 of DDR4. SLR1 has one interface to the bank #1 of DDR4. SLR2 does not have any memory interfaces. Placing instances of kernels on the same SLR that contains the memory interfaces used by the kernel doe not require any SLR crossings. SLR crossings use the limited SLL routes which are more expensive than the standard intra-SLR routing [147]. When SLR crossings are unavoidable, instances should be placed on an SLR that reduces the length and number of these crossings as much as possible. For a clock frequency of 450 MHz, the DDR4 memory has a total throughput of 36 GB/s, while the HBM2's total throughput is 425 MHz [132].

The AXI interface has independent reading and writing channels. Read and writes requests are queued in the memory controllers as only one command can be serviced at one time. The AXI memory protocol used to interface the FPGA chip with DDR4 and HBM2 memories requires linear (consecutive) access to enable high performance [144].

### 5.2.3 Xilinx Vitis High-Level Synthesis

Xilinx FPGA kernels can be written using Vitis HLS tool which enables converting C, C++ and OpenCL codes to SystemVerilog HDL RTL [148, 147]. Examples of projects using Vitis HLS include building communication primitives for a TCP/IP network stack using HLS [64] and developing HBM Connect an FPGA to HBM interface written in HLS that increases the memory bandwidth [39]. A Sobel algorithm accelerator design using HLS OpenCL was explored in [97]. A proposed probabilistic model to the processes scheduler allows it to explore further scheduling approaches [36]. An assessment of Vitis HLS in terms of ease of programming and profiling accelerators using Himeno Benchmark is done in [29].

The evaluation in [29] demonstrated that programmer's understanding of how to transform the code into a dataflow style, usage of Xilinx' HLS IDE, and deep knowledge of the board's hardware were necessary to develop efficient accelerators. One of the objective of this paper is to also experimentally evaluate writing a large application entirely in HLS while incorporating APIs from the new Vitis libraries. In addition, this work provides an opportunity to evaluate the capabilities of the Alveo U280 board.

### 5.2.4 Xilinx Vitis Sorting Libraries

The Vitis library includes four sorting APIs written in Xilinx HLS C/C++. The version used in this work is v2020.2. The sorting APIs are:

- Bitonic-Sort [139]: Sorting network that is highly parallelizable. It has time complexity of $\mathcal{O}(n\log(n)^2)$. Bitonic-Sort provides very high throughput, but consumes large amount of FPGA resources. As the number of keys double, resources needed for the implementation quadruple. Experimental results show that the high resources utilization for a number of keys higher than 128 is inconceivable.

- Insert-Sort [140]: In-place sorting algorithm. It is Highly efficient for sorting small sets with a time complexity of $\mathcal{O}(nk)$. $k$ is the maximum number of steps, any key must move to reach its correct position. The hardware implementation is resource efficient for maximum sort size of 1024 keys. The size of the internal shift register array is limited to 1k as this is the maximum size for array partitioning in HLS.

- Merge-Sort [141]: Merges two sorted streams of keys into one sorted stream. It has a time complexity of $\mathcal{O}(n \log(n))$.
- Compound-Sort [142]: Combines insert and merge sort implementations in one API. It consists of one insert sort module and multiple merge sort modules. Merge sort is used to merge the sorted 1k sequences produced by the insert sort module. The hardware implementation supports a maximum sorting size of 2M keys.

Evaluating the Vitis sorting routines showed that compound sort is the most suitable in terms supporting a large sorting size, while having a acceptable resources utilization. Its implementation is discussed further in Section 5.3.1.

## 5.3 Sample Sort Accelerator Design

The FPGA implementation of Sample sort accelerator is divided into four kernels: Kernel A sorts the sub-arrays and building the histogram matrix. Kernel B implements the transpose of the histogram matrix and the prefix-sums calculations. Kernel C implements the distribution of the keys from each sub-array into their corresponding buckets. Kernel D sorts the buckets.

### 5.3.1 Sorting the Sub-Arrays

Kernel A (Fig. 5.3) sorts the initial sub-arrays using the Vitis library compound sort (module E, Fig. 5.4). Module E uses two sorting sub-routines: insert sort and merge sort. The compound sort library can sort up to 1M keys [142]. However, due to the its very high resources

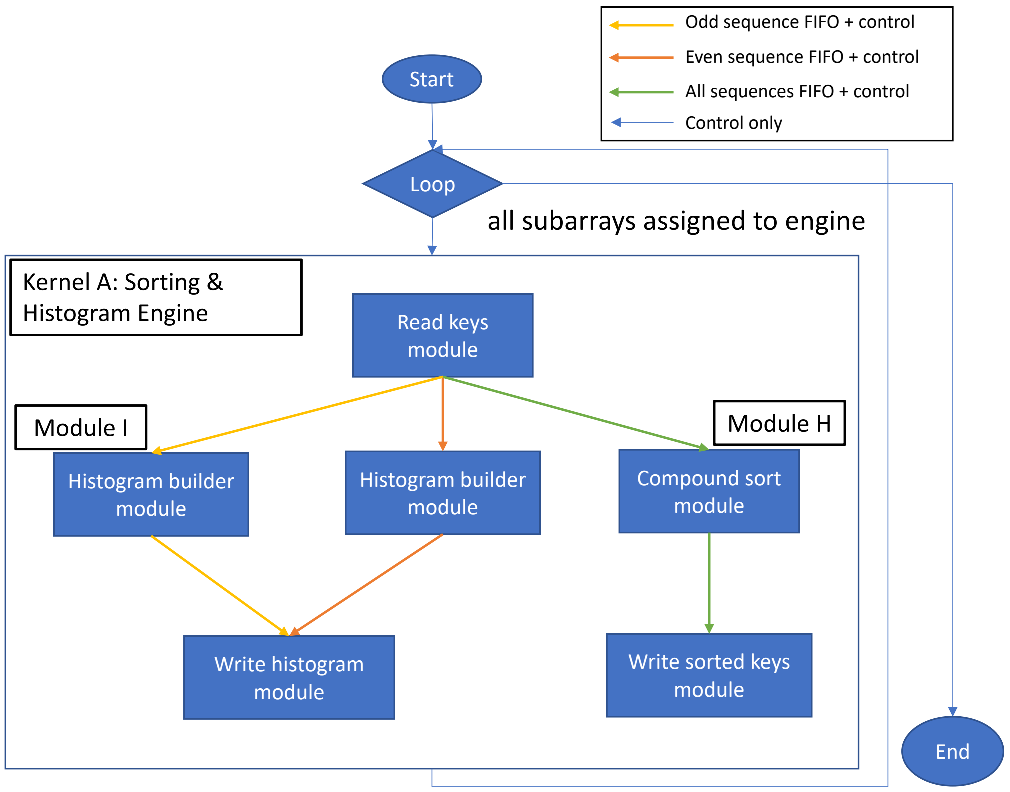


Figure 5.3: Kernel A - Each loop iteration sorts one sub-array and builds its histogram: a copy of each key is sent to the compound sort module (E) and to the histogram module (I).

utilization, it is not practical to implement an instance of module E with an array larger than 512K keys. In our implementation, module E is configured to support a sorting length of 256K keys.

Module E, uses the dataflow optimization to create concurrent processes [148], it is composed of modules F & G both executing in a dataflow fashion.

Module F (Fig. 5.5a) sorts and stores each 8K keys on-chip using URAM memory. The is done by using a single instance of insert sort module that sorts each 1K keys and passes that sorted sequence to the merge wrapper (module H, Figure 5.6) that contains seven instances of the merge sort module. Module H receives a sorted 1K sequence and pushes it in the next available FIFO.

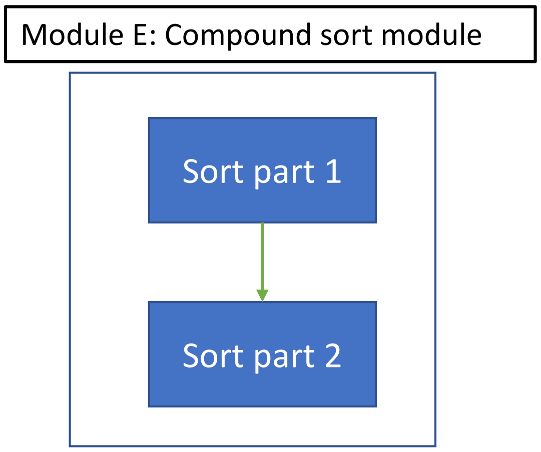


Figure 5.4: Module E - The Vitis compound sort module is composed of two modules: F: sorts each 8K keys and stores them on FPGA's URAM. G: merges all sorted 8K stream int one sorted stream.

The output of the final merge sort instance in module H is a sorted 8K sequence that is stored in a partitioned array of on-chip URAM. The URAM array is divided into 32 partitions, each partition can store up to 8K of sorted keys.

Module G (Fig. 5.5b), is composed of five merge wrapper instances. Keys are read from each of the 32 URAM memory partitions into the 32 inputs merge sort wrapper instance. The first merge wrapper instance, merges 32 streams of keys into 16 sorted streams. The final merge wrapper instance produces the sorted stream of keys. Sorted keys are written back to the same memory location in HBM2 replacing the unsorted sub-array with the sorted one.

Both read and write AXI interfaces in this design have the maximum data-width size of 512 bits since the loop sizes are known at synthesis time. Each instance of kernel A can be connected to one or more HBM2 banks. A single AXI memory channel is synthesized to connect the kernel to four HBM2 banks. Thus, enabling each kernel to sort 1GB of data i.e. 256M of 4-byte keys.

### 5.3.2 Building Histogram

The histogram building module (module I, Fig. 5.7) executes concurrently with module E; it has an Initiation Interval (II) of two cycles to aid the design in meeting timing, hence two instances of module I are instantiated in kernel A. The histogram matrix is built using sets of registers. The AXI write interface uses the maximum data-width size of 512 bits.

### 5.3.3 Matrix Transpose & Prefix-Sum

Kernel B (Fig. 5.8) transposes the histogram matrix and builds a number of the prefix-sums that is equal to the number of parallel write channels available to the FPGA. The histogram matrix is read and each bin is stored in the transposed order in the on-chip memory. Random access to on-chip memory does not incur any additional delays, thus enabling efficient writing of the histogram in a transposed order. The parallel distribution of the buckets (kernel C) requires a prefix-sum for each parallel partial bucket writing process. For 16 parallel distribution processes, 16 sequences of prefix-sums are computed in a pipelined fashion and written back to the HBM2 memory. Both read and write AXI interfaces have the maximum data-width size of 512 bits.

### 5.3.4 Distribution of Keys

Kernel C (Fig. 5.9) distributes keys from each sub-array to their corresponding buckets. The kernel reads and distributes keys from one sub-array at a time to partial buckets stored in a mix of BRAM and URAM memories. When the entire sub-array has been processed, its partial buckets are ready to be written back to a new locations on the HBM2 memory. Writing all partial buckets serially to a single HBM2 bank would prevent any opportunity for parallelism. Alternatively, issuing parallel writes to multiple memory channels to the same bank would divide the available write bandwidth by the number of parallel write requests. Parallelism in kernel C is made possible by connecting it to 16 HBM2 banks with 16 parallel processes reading 16 partial buckets from on-chip memory and writing them simultaneously to 16 HBM2 banks. The limit on the number of parallel write processes is dependent the available number of HBM2 memory banks and the resources needed for implementing the channels to each bank. A vector of the buckets sizes, computed in kernel C, is also written back to HBM2 memory and used by Kernel D. The read AXI interface has the maximum data-width size of 512 bits, while the write interface is 32 bits due to the variable sizes of the partial buckets.

### 5.3.5 Sorting The Buckets

Kernel D (Fig. 5.10) sorts the buckets. This kernels relies on module E. However, since the buckets have a variable size, module E is configured to support the largest possible size of a bucket for a given distribution of keys. Hence, the data-width for reads and writes AXI interfaces is 32 bits.

## 5.4 Experimental Evaluation

In this design, it is not possible to use all three SLRs at the same time since only SLR0 has an interface to the HBM2 memory banks. Therefore, placing instances of kernels on SLR1 & SLR2 causes routing conflicts due to the scarcity of the SLL routing resources (discussed in Section 5.2.2). Hence, the FPGA is reconfigured three times during the sample sort implementation: the

FPGA is initially configured with instances of kernel A, then with instances of kernels B & C, and finally with instances of kernel D.

### 5.4.1 Kernel A

Four configurations of kernel A were tested (Table 5.1). In all four configurations, the sub-arrays in the dataset are divided such that each sub-array has 256K keys (1 MB). Module E (compound sort) in kernel A is configured to support a maximum sorting size of 256K of 32-bit keys. Resources utilization, clock frequency, kernel's latency and throughput for all configurations are shown in Table 5.2.

Table 5.1: Specifications of kernel A configurations

| | Instances # | Banks per instance # |
|---|---:|---:|
| Config 1 | 1 | 2 |
| Config 2 | 1 | 4 |
| Config 3 | 2 | 2 |
| Config 4 | 2 | 4 |

In configurations 1 and 2 have a single instance of kernel A, while in configurations 2 and 4 have two instances. In configurations 1 and 3, an instance is connected to two HBM2 banks (512 MB) using a single AXI channel. Thus, an instance is sorting and building the histogram for 512 sub-arrays. In configurations 2 and 4, each instance is connected to four HBM2 banks (1 GB) using a single AXI channel. Therefore, an instance is sorting and building the histogram for 1K sub-arrays.

Table 5.2: Resources Utilization for four configurations of kernel A on SLR0

| | **CLB [%]** | **CLB LUTs [%]** | **CLB Reg [%]** | **BRAM [%]** | **URAM [%]** | **Clock Freq. [MHz]** | **Latency [ms]** | **Throughput [MB/sec]** |
|---|---|---|---|---|---|---|---|---|
| **Config 1** | 54.16 | 26.15 | 16.12 | 13.39 | 10 | 298 | 925 | 540 |
| **Config 2** | 65.89 | 30.53 | 16.86 | 13.39 | 10 | 232 | 2,375 | 420 |
| **Config 3** | 88.97 | 48.87 | 27.96 | 22.69 | 20 | 247 | 1,114 | 900 |
| **Config 4** | 95.48 | 55.01 | 35.20 | 22.40 | 20 | 223 | 2,483 | 800 |

Table 5.2 shows that connecting two more HBM2 banks to an instance slightly increases the total CLB utilization while significantly decreasing the achievable clock frequency of the kernel. URAM utilization, used to store the sorted 8K sequences, changes linearly with the number of instances. The very high CLB utilization in configurations 3 and 4 mainly is attributed to module E. With an average of 90% CLB utilization, no more instances or any other user's logic can be placed and routed on SLR0. Because of this high CLB utilization, placing instances of kernel A on SLR1 or SLR2 was not feasible as the routing from these SLRs to the HBM2 interface on SLR0 becomes impossible. Configuration 3 provides the highest throughput for this step in sample sort.

### 5.4.2 Kernels B & C

An instance of kernel B is implemented on SLR0, while an instance of kernel C is implemented on SLR1. Table 5.3 shows the resources utilization, clock frequency, kernel's latency and throughput for two configurations of kernel B. Configuration 1 uses on-chip registers and BRAM for storage of the transposed histogram matrix transpose, while configuration 2 relies on URAM resources for the storage. The higher storage capability of URAM allows for better utilization of the HBM2 bandwidth.

Table 5.3: Resources Utilization for kernel B on SLR0

| **Resources/ Config.** | **CLB [%]** | **CLB LUTs [%]** | **CLB Reg [%]** | **BRAM [%]** | **URAM [%]** | **Clock Freq. [MHz]** | **Latency [ms]** | **Throughput [MB/sec]** |
|---|---|---|---|---|---|---|---|---|
| **Config. 1** | 19.28 | 6.68 | 6.86 | 40.33 | 0 | 310 | 0.9 | 278 |
| **Config. 2** | 17.21 | 5.93 | 5.88 | 22.17 | 80 | 272 | 12 | 333 |

Kernel C is implemented on SLR1 and utilizes both BRAM & URAM to to provide sufficient storage for the partial buckets before writing them to 16 HBM2 banks in parallel. In spite of the moderate utilization of CLBs, clock frequency is limited. This is due to the high utilization of both types of memory resources.

Table 5.4: Resources Utilization for kernel C on SLR1

| | **CLB [%]** | **CLB LUTs [%]** | **CLB Reg [%]** | **BRAM [%]** | **URAM [%]** | **Clock Freq. [MHz]** | **Latency [ms]** | **Throughput [MB/sec]** |
|---|---|---|---|---|---|---|---|---|
| **Utilization** | 41.95 | 17.47 | 14.21 | 81.03 | 80 | 224 | 739 | 347 |

### 5.4.3 Kernel D

Similarly to kernel A, four configurations of kernel D were tested (Table 5.5). However, Kernel D is sorting variable sized buckets and does not include any instances of module I (histogram building). Module E here is configured to support a maximum sorting size of 512K of 32-bit keys. Hence, sorting buckets with a size larger than that of a sub-array is supported. Resources utilization, clock frequency, kernel's latency and throughput for all configurations are shown in Table 5.6.

Table 5.5: Specifications of kernel D configurations

| | Instances # | Banks per instance # |
|---|---:|---:|
| Config 1 | 1 | 2 |
| Config 2 | 1 | 4 |
| Config 3 | 2 | 2 |
| Config 4 | 2 | 4 |

Table 5.2 shows better throughput in configurations of kernel D comparing to those of kernel A (Table 5.2). Comparing to kernel A, higher clock frequency is achievable in kernel D since no resource is utilized over 65%. The routing algorithm targets a preset high clock frequency and when it fails to meet timing, it automatically scales down the clock frequency to meet timing.

### 5.4.4 FPGA Reconfiguration

Since it is not possible to fit all kernels on the FPGA chip, reconfiguration between steps is needed. FPGA reconfiguration time ranges from 3 to 9 seconds depending on the size of the bit-

Table 5.6: Resources Utilization for four configurations of kernel D on SLR0

| | CLB [%] | CLB LUTs [%] | CLB Reg [%] | BRAM [%] | URAM [%] | Clock Freq. [MHz] | Latency [ms] | Throughput [MB/sec] |
|---|---|---|---|---|---|---|---|---|
| **Config 1** | 36.26 | 23.45 | 12.39 | 6.47 | 20 | 313 | 894 | 573 |
| **Config 2** | 37.03 | 23.26 | 12.38 | 6.47 | 20 | 300 | 2,217 | 461 |
| **Config 3** | 63.72 | 43.17 | 20.60 | 9.45 | 40 | 294 | 956 | 1,071 |
| **Config 4** | 63.73 | 43.18 | 20.59 | 9.45 | 40 | 289 | 1,945 | 1,053 |

stream. Reconfiguring the FPGA chip on the Xilinx Alveo platforms causes all data on the board's memory to be lost. Hence, data must be moved back to the host after all instances complete running and copied again to the HBM2 banks after reconfiguration of the new instances is complete. The need to copy data from off-chip memory back and forth for FPGA reconfiguration is a limitation that should be mitigated in future platforms. The write bandwidth from host to HBM2 is 10 GB/sec, while the read bandwidth from HBM2 to host is 9 GB/sec. Table 5.7 shows a breakdown of the latency for data movement, kernel configuration and execution for a 1 GB dataset. We can see that while the data transfer time between the host and the FPGA is relatively small, the reconfiguration time is more than twice the total execution times of all four kernels.

### 5.4.5 Throughput comparison

A parallel software implementation of sample sort included in ParlayLib [6] is used to evaluate the performance of the proposed sample sort accelerator. ParlayLib is a lightweight C++

Table 5.7: Latency breakdown for each step of the execution of sample sort accelerator for a dataset of 1 GB

| | **Write host to HBM2 [ms]** | **Reconfigure FPGA [ms]** | **Kernel Exe. Time [ms]** | **Read HBM2 to host [ms]** |
|---|---:|---:|---:|---:|
| **Kernel A** | 134 | 3,000 | 1,114 | 122 |
| **Kernels B & C** | 135 | 9,000 | 2,963 | 123 |
| **Kernel D** | 134 | 3,00 | 956 | 122 |
| **Total** | 403 | 12,300 | 5,033 | 367 |

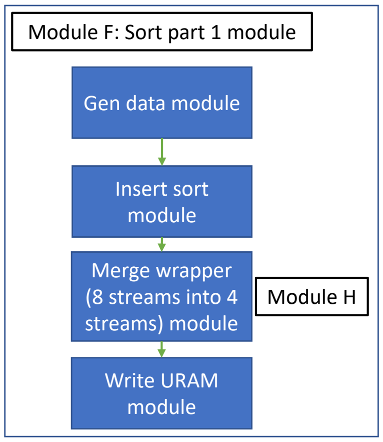
header-only library that can be used for developing parallel applications on shared-memory multi-core machines. The CPU used to run the software implementation is an Intel Xeon CPU E5-2640 v3 clocked at 2.6 GHz. It has two sockets, eight cores per socket and two threads per core. The L1 data and instruction caches is 32KB for each. L2 cache is 256KB and L3 cache is 20MB. Two configurations of the CPU are used. A configuration with 32 cores (32 threads) and another configuration uses eight cores (eight physical cores). Table 5.8 show the latency, throughput and throughput per core (for CPU) and per engine (for FPGA). The throughput of a single FPGA engine is 5.7x higher than that of a CPU core. When we get FPGAs that are large enough to fit everything, not counting the change in clock frequency it will be furthermore competitive.

Table 5.8: Latency & throughput comparison between various platforms and sorting algorithms. Dataset is 256M recordsa.

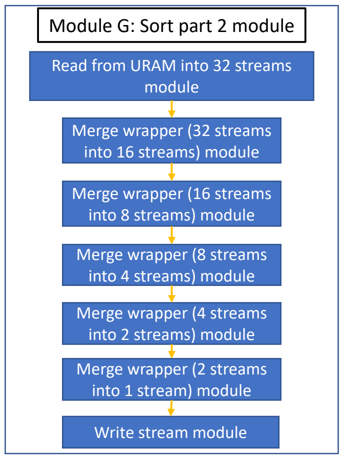
| **Platform** | **Latency [sec]** | **Throughput [Million Rec/sec]** | **Throughput per eng./core [Million Rec/sec]** |
|---|---|---|---|
| **FPGA (2 engines)** | 5 | 54 | 27 |
| **CPU (32 cores)** | 1.8 | 149 | 4.7 |
| **CPU (8 cores)** | 3.5 | 77 | 9.6 |

## 5.5 Conclusion

In this chapter, a novel parallel sample sort has been presented. Sample sort is a cache-oblivious sorting algorithm that enables sorting large datatsets using the limited local memory. The implementation is done entirely using Vitis HLS. The design explore the capabilities of Vitis HLS and the Xilinx Alveo U280 board to build a none-trivial application. The design is divided into four type of kernels that are reconfigured in three steps in order to accommodate their placement to be closer to the global memory resources. Experimental evaluation comparing the FPGA throughput to that of a parallel software implementation on CPU shows that the throughput of a single FPGA engine is 5.7x higher than that of a CPU core.


Module F: Sort part 1 module
Gen data module
Insert sort module
Merge wrapper (8 streams into 4 streams) module
Module H
Write URAM module

(a) Module F - Gen data module counts the number of keys in the stream. Insert sort module sorts each stream of 1K keys. The merge wrapper module merges every eight sorted 1K streams into a 8K stream. The final module stores each sorted 8K stream on a URAM partition.


Module G: Sort part 2 module
Read from URAM into 32 streams module
Merge wrapper (32 streams into 16 streams) module
Merge wrapper (16 streams into 8 streams) module
Merge wrapper (8 streams into 4 streams) module
Merge wrapper (4 streams into 2 streams) module
Merge wrapper (2 streams into 1 stream) module
Write stream module

(b) Module G - The first module reads sorted keys from the 32 URAM partitions and distributes them to the 32 inputs of the next merge instance. All 32 streams are merged to produce the final sorted sequence.

Figure 5.5: Modules F & G

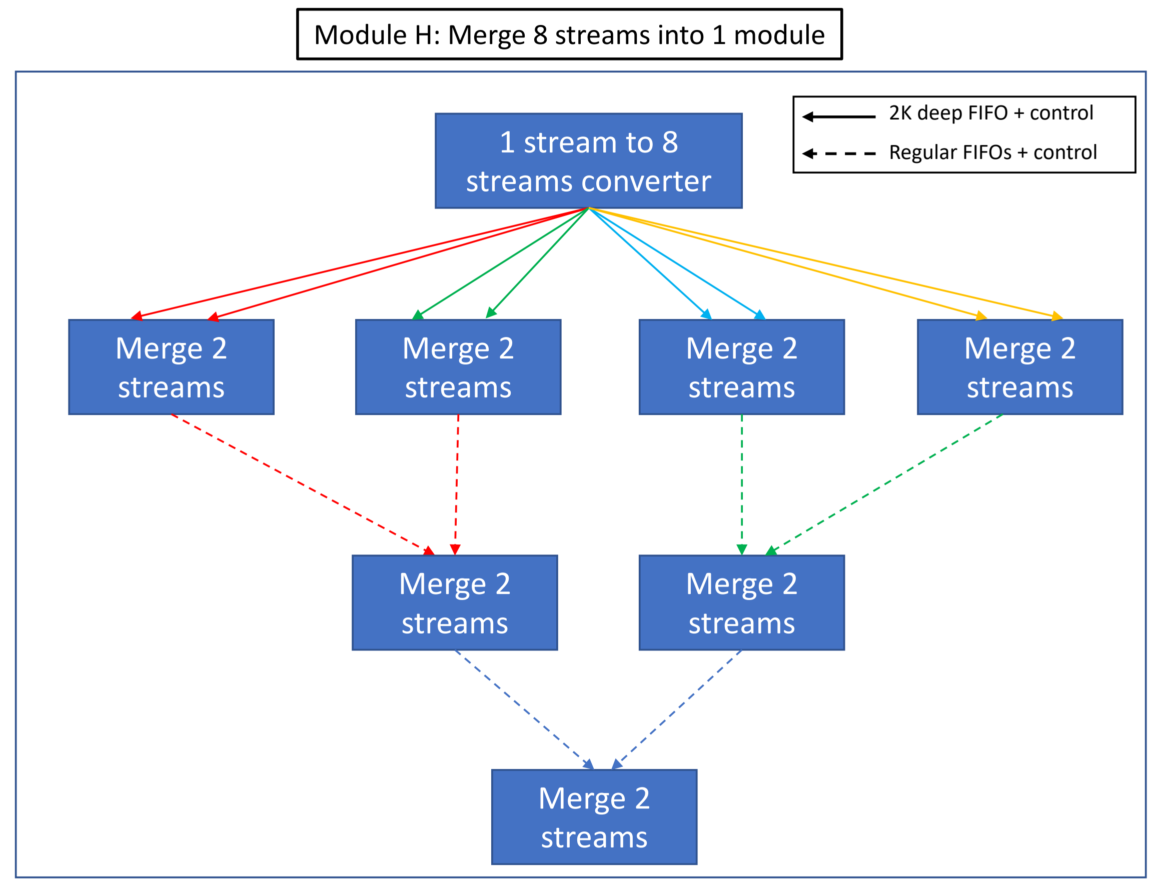


Figure 5.6: Module H - The merge sort wrapper is made up of seven merge sort modules. The converter module receives sorted keys and stores each 1K stream in the next available FIFO. Eight FIFOs, each 2K deep, prevent a stalling of pipeline. This module produces a stream of 8K sorted keys.

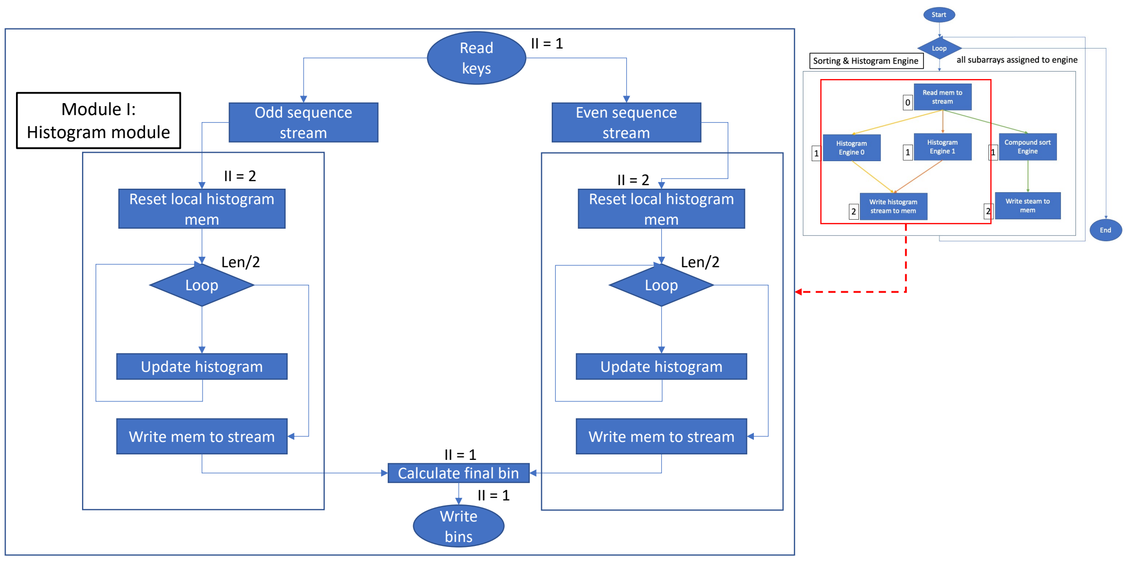


Figure 5.7: Module I - The histogram building module within kernel A has an initiation interval of 2. Two instances of module I are simultaneously active insuring full throughput.

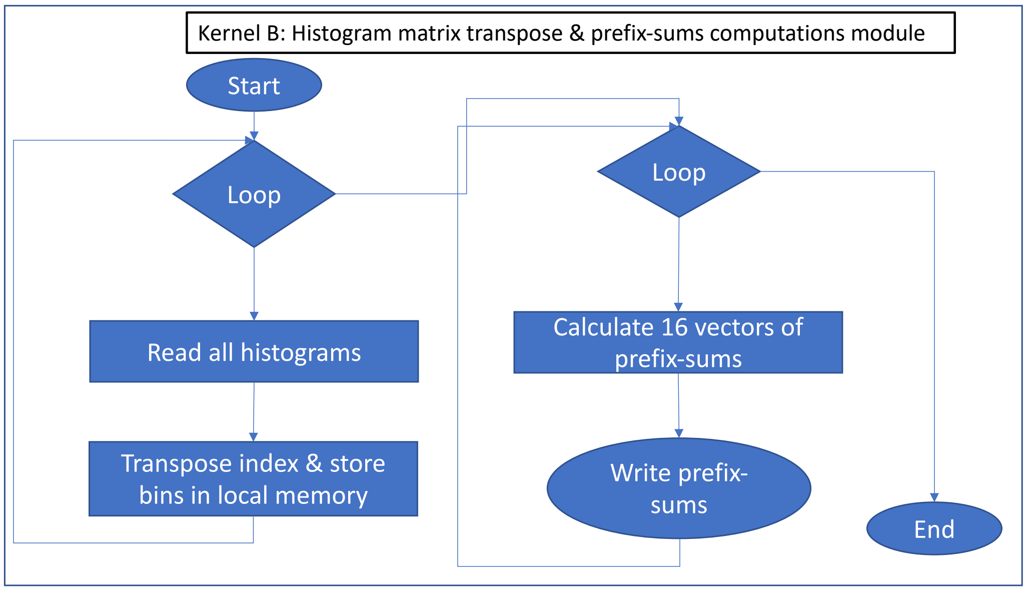

Kernel B: Histogram matrix transpose & prefix-sums computations module
Start
Loop
Read all histograms
Transpose index & store bins in local memory
Loop
Calculate 16 vectors of prefix-sums
Write prefix-sums
End

Figure 5.8: Kernel B: Transposing the histogram matrix is done on the fly as bins are being read. The entire transposed histogram matrix fits on local memory. Each bin that has been read is stored in its new location on on-chip memory. 16 sequences of prefix-sum are needed for parallel writing of 16 buckets in the keys distribution step. Prefix-Sums bins are sent to writing FIFO as soon as they are computed.

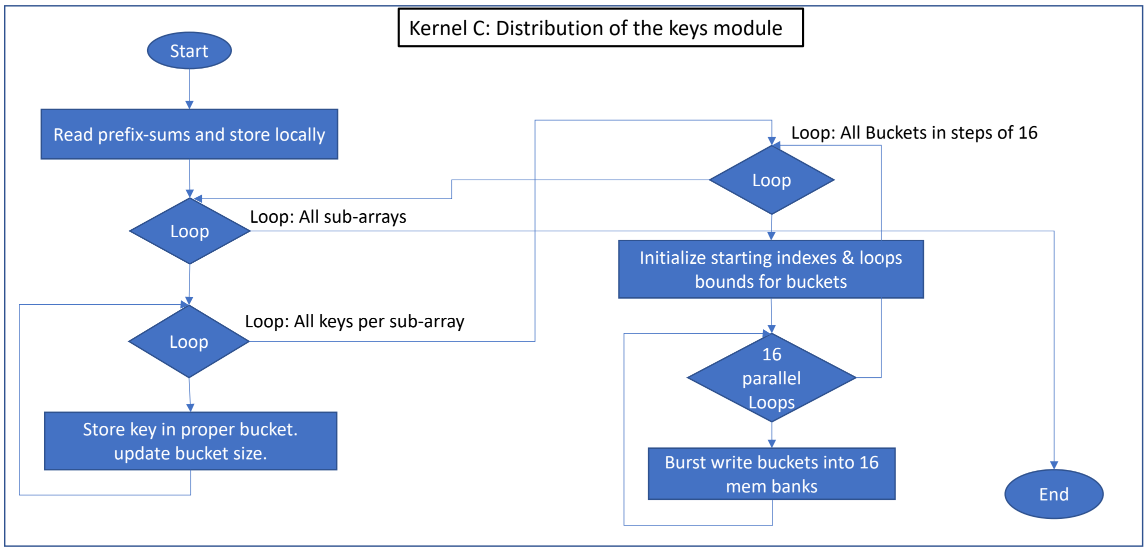

Kernel C: Distribution of the keys module
Start
Read prefix-sums and store locally
Loop
Loop: All sub-arrays
Loop
Loop: All keys per sub-array
Store key in proper bucket. update bucket size.
Loop: All Buckets in steps of 16
Loop
Initialize starting indexes & loops bounds for buckets
16 parallel Loops
Burst write buckets into 16 mem banks
End

Figure 5.9: Kernel C - Parallel distribution of keys. Keys from each sub-array, opne per iteration, are read and distributed to partial buckets stored on chip using a mix of BRAM & URAM. Two loops read keys in parallel from 16 partial buckets and write the partial buckets in parallel to 16 HBM2 banks.

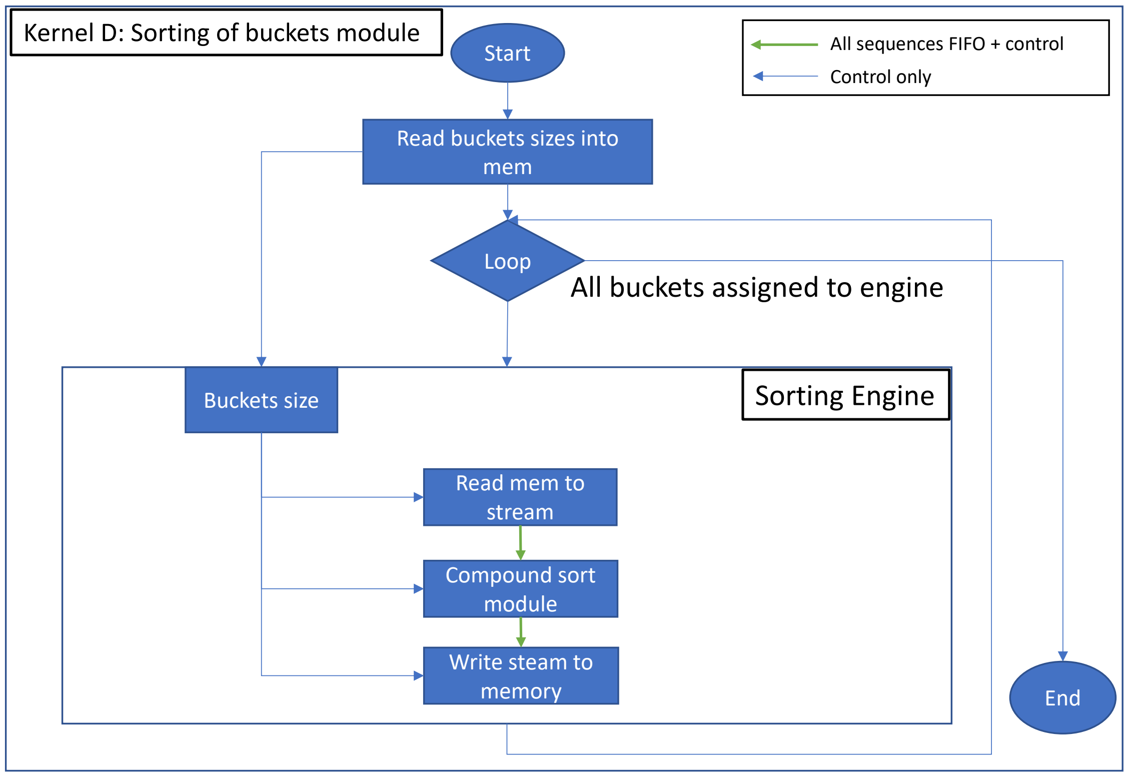

Kernel D: Sorting of buckets module
Start
All sequences FIFO + control
Control only
Read buckets sizes into mem
Loop
All buckets assigned to engine
Buckets size
Sorting Engine
Read mem to stream
Compound sort module
Write steam to memory
End

Figure 5.10: Kernel D sorts the buckets. Module E is the main component in Kernel D. The sorting size of each bucket is required and read from HM2.

# Chapter 6

# Conclusions

Driven by the decreasing cost of DRAM, in-memory processing is becoming more common to store and process large datasets. This dissertation explores FPGA-based in-memory acceleration for two database primitives which are group-by aggregation and sorting.

In Chapter 3, a hardware multithreading approach has been explored for the hash-based group-by aggregation. Group-by aggregation, do not exhibit control flow or data localities. To overcome this issue, latency masking is achieved by issuing hundreds of hardware threads across four FPGAs. The FGL MTP design presented in this dissertation reduces contention among locks and enables higher parallelism and throughput that is 1.6X to 2.1X times better than the previous work. The FGL MTP design is compared against five software approaches (both partitioned, and non-partitioned) over five different datasets. Unlike the software approaches, the FGL MTP design's throughput is unaffected by the benchmark's cardinality achieving an average of 3.3x speedup over all CPU implementations.

In chapter 4, a novel in-memory parallel implementation of radix sort HARS is presented. HARS utilizes the large on-chip memory available on the new the Xilinx UltraScale+ series to store the histograms on the FPGA's local memory. This design significantly reduces the number of memory requests between the FPGA and off-chip memory. Thus, improving the sorting throughput. HARS achieves a throughput of 44 Million 128-bits records per second and is 1.4x faster than the CPU. Te raw throughput for GPU is about twice that of HARS due the GPU's much higher memory bandwidth. When normalising the throughput by the platform's available memory bandwidth, HARS achieves 1.36x higher throughput.

Finally, in chapter 5, a novel sample sort which is a cache-oblivious sorting algorithm. Sample sort enables sorting very large datasets using the platform's local memory by dividing the sorting steps and the dataset into operations on data small enough to fit on the FPGA's local memory. The design presented in this dissertation is done completely using Vitis HLS. The design is divided into four type of kernels that are reconfigured in three steps in order to accommodate their placement to be closer to the global memory resources. The design enables sorting large datasets without the need for large merge steps used in traditional sorting methods. Experimental evaluation comparing the throughput a parallel software implementation of sample sort on CPU to the FPGA implementation presented in this dissertation shows that the throughput of a single FPGA engine is 5.7x higher than that of a single CPU core.

# Bibliography

[1] Baidu. 2016.

[2] Intel agilex I-Series SoC FPGA product table. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/intel-agilex-i-series-product-table.pdf. Accessed: 2020-1-12.

[3] Intel stratix 10 dx product table. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-dx-product-table.pdf. Accessed: 2020-1-12.

[4] Intel® xeon® processor E5-2640 v3 product specifications. https://ark.intel.com/content/www/us/en/ark/products/83359/intel-xeon-processor-e5-2640-v3-20m-cache-2-60-ghz.html. Accessed: 2020-1-12.

[5] NVIDIA TITAN X pascal specs. https://www.techpowerup.com/gpu-specs/titan-x-pascal.c2863. Accessed: 2020-1-8.

[6] parlaylib: ParlayLib - a toolkit for programming parallel algorithms on Shared-Memory multicore machines.

[7] SDAccel development environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html. Accessed: 2021-7-23.

[8] std::sort - cppreference.com. https://en.cppreference.com/w/cpp/algorithm/sort. Accessed: 2019-12-23.

[9] UltraRAM: Breakthrough embedded memory integration on UltraScale+ devices (WP477).

[10] UltraScale architecture and product data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. Accessed: 2020-1-12.

[11] Ildar Absalyamov, Prerna Budhkar, Skyler Windh, Robert J. Halstead, Walid A. Najjar, and Vassilis J. Tsotras. FPGA-accelerated Group-by Aggregation Using Synchronizing Caches. In *Proc. of the 12th Int. Workshop on Data Management on New Hardware*, DaMoN '16, pages 11:1–11:9, NY, USA, 2016. ACM.

[12] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *Proc. of the VLDB Endowment*, 5(10):1064–1075, 2012.

[13] Alpha Data. http://www.alpha-data.com/dcp/capi.php, 2015.

[14] Oriol Arcas-Abella, Adrià Armejach, Timothy Hayes, Gorker Alp Malazgirt, Oscar Palomar, Behzad Salami, and Nehir Sonmez. Hardware acceleration for query processing: Leveraging FPGAs, CPUs, and memory. *Comput. Sci. Eng.*, 18(1):80–87, January 2016.

[15] Dmitri I Arkhipov, Di Wu, Keqin Li, and Amelia C Regan. Sorting with GPUs: A survey. September 2017.

[16] AWS Events. AWS re:invent 2019: Amazon redshift reimagined: RA3 and AQUA (ANT230). `https://youtu.be/6pZrE_tveLI`, December 2019. Accessed: 2020-7-11.

[17] R Ayoubi, S Istambouli, A Abbas, and G Akkad. Hardware architecture for a Shift-Based parallel Odd-Even transposition sorting network. In *2019 Fourth International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)*, pages 1–6, July 2019.

[18] C. Balkesen, J. Teubner, G. Alonso, and M.T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proc. of the 2013 IEEE Int. Conf. on Data Engineering*, ICDE'13, pages 362–373, 2013.

[19] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. of the VLDB Endowment*, 7(1):85–96, 2013.

[20] Nagender Bandi, Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Fast Data Stream Algorithms Using Associative Memories. In *Proc. of the 2007 ACM SIGMOD Int. Conf. on Management of Data*, SIGMOD '07, pages 247–256, 2007.

[21] R Barber, G Lohman, V Raman, R Sidle, S Lightstone, and B Schiefer. In-memory BLU acceleration in IBM's DB2 and dashDB: Optimized for modern workloads and hardware architectures. In *2015 IEEE 31st Int. Conf. on Data Engineering*, pages 1246–1252. ieeexplore.ieee.org, April 2015.

[22] Bashar Romanous, Mohammadreza Rezvani, Junjie Huang, Daniel Wong, Evangelos E. Papalexakis, Vassilis J. Tsotras, and Walid Najjar. High-Performance parallel radix sort on FPGA. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, volume 0, pages 224–224, May 2020.

[23] Yomna Ben Jmaa, Rabie Ben Atitallah, David Duvivier, and Maher Ben Jemaa. A comparative study of sorting algorithms with FPGA acceleration by high level synthesis. *Computación y Sistemas*, 23(1):213, March 2019.

[24] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Proc. of the 2011 ACM SIGMOD Int. Conf. on Management of Data*, SIGMOD'11, pages 37–48, 2011.

[25] Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 189–199, New York, NY, USA, June 2010. Association for Computing Machinery.

[26] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. of the 25th Int. Conf. on Very Large Data Bases*, VLDB'99, pages 54–65, 1999.

[27] Jeremy Branscome, Michael Corwin, Liuxi Yang, James Shau, Ravi Krishnamurthy, and Joseph I. Chamdani. Processing elements of a hardware accelerated reconfigurable processor for accelerating database operations and queries. Patent: US 20080189251 A1, 2008.

[28] Technical Brief. TeraSort benchmark comparison for YARN. `https://mapr.com/whitepapers/terasort-benchmark-comparison-yarn/assets/terasort-comparison-yarn.pdf`.

[29] N Brown. Weighing up the new kid on the block: Impressions of using vitis for HPC software development. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 335–340, August 2020.

[30] Prerna Budhkar, Ildar Absalyamov, Vasileios Zois, Skyler Windh, Walid A. Najjar, and Vassilis J. Tsotras. Accelerating in-memory database selections using latency masking hardware threads. *ACM Trans. Archit. Code Optim.*, 16(2):13:1–13:28, April 2019.

[31] M Burtscher, R Nasre, and K Pingali. A quantitative study of irregular programs on GPUs. In *2012 IEEE Int. Symp. on Workload Characterization (IISWC)*, pages 141–151, November 2012.

[32] Jared Casper and Kunle Olukotun. Hardware Acceleration of Database Operations. In *Proc. of the 2014 ACM/SIGDA Int. Symp. on Field-programmable Gate Arrays*, pages 151–160, 2014.

[33] CedermanDaniel and TsigasPhilippas. GPU-Quicksort. *ACM J. Exp. Algorithmics*, January 2010.

[34] Ren Chen, Sruja Siriyal, and Viktor Prasanna. Energy and memory efficient mapping of bitonic sorting on FPGA. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 240–249, New York, NY, USA, February 2015. Association for Computing Machinery.

[35] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3), August 2007.

[36] Jianyi Cheng, John Wickerson, and George A Constantinides. Probabilistic scheduling in High-Level synthesis. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 195–203. ieeexplore.ieee.org, May 2021.

[37] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. A study of main-memory hash joins on many-core processor: A case with intel knights landing architecture. In *Proc. of the 2017 ACM on Conf. on Information and Knowledge Management*, CIKM '17, pages 657–666, NY, USA, 2017. ACM.

[38] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings VLDB Endowment*, 1(2):1313–1324, August 2008.

[39] Young-Kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. HBM connect: High-Performance HLS interconnect for FPGA HBM. *FPGA*, 2021:116–126, February 2021.

[40] John Cieslewicz and Kenneth A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *Int. Conf. on Very Large Data Bases*, VLDB '07, pages 339–350, 2007.

[41] John Cieslewicz and Kenneth A. Ross. Data Partitioning on Chip Multiprocessors. In *Proc. of the 4th Int. Workshop on Data Management on New Hardware*, DaMoN '08, pages 25–34, 2008.

[42] Convey Computers. http://www.conveycomputer.com, 2015.

[43] A Davidson, D Tarjan, M Garland, and J D Owens. Efficient parallel merge sort for fixed and variable length keys. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, May 2012.

[44] Tony Baer (dbInsight). Amazon redshift turns AQUA. `https://www.zdnet.com/article/amazon-redshift-turns-aqua/`. Accessed: 2020-8-17.

[45] Udit Dhawan and André Dehon. Area-Efficient Near-Associative Memories on FPGAs. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 7(4):1–22, January 2015.

[46] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proc. of the ACM Int. Conf. on Management of Data*, SIGMOD '13, pages 1243–1254, NY, USA, 2013. ACM.

[47] Jian Fang, Yvo T B Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. In-memory database acceleration on FPGAs: a survey. *VLDB J.*, October 2019.

[48] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database–An architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[49] A Farmahini-Farahani, H J Duwe, III, M J Schulte, and K Compton. Modular design of High-Throughput, Low-Latency sorting units. *IEEE Trans. Comput.*, 62(7):1389–1402, July 2013.

[50] A Farmahini-Farahani, A Gregerson, M Schulte, and K Compton. Modular high-throughput and low-latency sorting units for FPGAs in the large hadron collider. In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 38–45, June 2011.

[51] E B Fernandez, W A Najjar, S Lonardi, and J Villarreal. Multithreaded FPGA acceleration of DNA sequence mapping. In *2012 IEEE Conf. on High Performance Extreme Computing*, pages 1–6. ieeexplore.ieee.org, September 2012.

[52] Phil Francisco. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics, 2011.

[53] W D Frazer and A C McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, July 1970.

[54] M Frigo, C E Leiserson, H Prokop, and S Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 285–297. ieeexplore.ieee.org, October 1999.

[55] Michael Gowanlock and Ben Karsin. A hybrid CPU/GPU approach for optimizing sorting throughput. *Parallel Comput.*, 85:45–55, July 2019.

[56] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter Weinberger. Quickly Generating Billion-record Synthetic Databases. In *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data*, SIGMOD'94, pages 243–252, 1994.

[57] Greenplum. http://www.pivotal.io/big-data/pivotal-greenplum-database, 2015.

[58] Cristian Grozea, Zorana Bankovic, and Pavel Laskov. FPGA vs. multi-core CPUs vs. GPUs: Hands-On experience with a sorting application. In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore-Challenge: Aspects of New Paradigms and Technologies in Parallel Computing*, pages 105–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[59] P K Gupta. Accelerating datacenter workloads. In *26th Int. Conf. on Field Programmable Logic and Applications (FPL)*. fpl2016.org, 2016.

[60] Robert J. Halstead, Ildar Absalyamov, Walid A. Najjar, and Vassilis J. Tsotras. FPGA-based Multithreading for In-Memory Hash Joins. In *CIDR 2015, Seventh Biennial Conf. on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proc.*, CIDR'15, 2015.

[61] Robert J Halstead, Walid A Najjar, and Omar Huseini. SpVM acceleration with latency masking threads on FPGAs. *Algorithms*, 20:21, 2014.

[62] Robert J. Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh Asaad, and Balakrishna Iyer. Accelerating Join Operation for Relational Databases with FPGAs. In *Proc. of the 2013 IEEE 21st Ann. Int. Symp. on Field-Programmable Custom Computing Machines*, FCCM'13, pages 17–20, 2013.

[63] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, December 2009.

[64] Zhenhao He, Dario Korolija, and Gustavo Alonso. EasyNet: 100 gbps network for HLS. In *International Conference on Field-Programmable Logic and Applications (FPL 2021)*. research-collection.ethz.ch, 2021.

[65] Foster D. Hinshaw, David L. Meyers, and Barry M. Zane. Programmable streaming data processor for database appliance having multiple processing unit groups. Patent: US 7577667 B2, 2009.

[66] IBM Netezza. www.ibm.com/software/data/netezza/, 2014.

[67] F IlyasIhab, BeskalesGeorge, and A SolimanMohamed. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, October 2008.

[68] Hiroshi Inoue and Kenjiro Taura. SIMD- and cache-friendly algorithm for sorting an array of structures. *Proceedings VLDB Endowment*, 8(11):1274–1285, July 2015.

[69] M.F. Ionescu and K.E. Schauser. Optimizing Parallel Bitonic Sort. In *Proc. of the 11th Int. Symp. on Parallel Processing*, pages 303–309, 1997.

[70] J. Gray, C. Nyberg, M. Shah, and N. Govindaraju. Sort benchmark home page. `http://sortbenchmark.org/`, 2016. Accessed: 2019-12-12.

[71] S Jun and S Xu. Terabyte sort on FPGA-Accelerated flash storage. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–24. ieeexplore.ieee.org, April 2017.

[72] K. E. Batcher Goodyear Aerospace Corporation, Akron, and Ohio. Sorting networks and their applications — proceedings of the april 30–may 2, 1968, spring joint computer conference. `https://dl.acm.org/doi/10.1145/1468075.1468121`. Accessed: 2020-1-12.

[73] Konstantinos Kalaitzis, Evripidis Sotiriadis, Ioannis Papaefstathiou, and Apostolos Dollas. Evaluation of external memory access performance on a High-End FPGA hybrid computer. *Computation*, 4(4):41, October 2016.

[74] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th Int. Conf. on Data Engineering*, pages 195–206. ieeexplore.ieee.org, 2011.

[75] Omar Khattab, Mohammad Hammoud, and Omar Shekfeh. Polyhj: A polymorphic main-memory hash join paradigm for multi-core machines. In *Proc. of the 27th ACM Int. Conf. on Information and Knowledge Management*, CIKM '18, pages 1323–1332, NY, USA, 2018. ACM.

[76] Kickfire. http://www.teradata.com/, 2015.

[77] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proc. of the VLDB Endowment*, 2(2):1378–1389, August 2009.

[78] Kangnyeon Kim, Ryan Johnson, and Ippokratis Pandis. BionicDB: Fast and power-efficient OLTP on FPGA. In *EDBT*, 2019.

[79] Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *Proc. VLDB Endowment*, 9(4):252–263, December 2015.

[80] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proc. of the 46th Ann. IEEE/ACM Int. Symp. on Microarchitecture*, MICRO-46, pages 468–479, NY, USA, 2013. ACM.

[81] Dirk Koch and Jim Torresen. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 45–54, New York, NY, USA, 2011. ACM.

[82] A. Krikelis and C. C. Weems. Associative processing and processors. *Computer*, 27(11):12–17, 1994.

[83] J. T. Kuehn and B. J. Smith. The Horizon Supercomputing System: Architecture and Software. In *Proc. of the 1988 ACM/IEEE Conf. on Supercomputing*, Supercomputing '88, pages 28–34, Los Alamitos, CA, USA, 1988. IEEE.

[84] M. Kumar and D.S. Hirschberg. An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and Its Application in Parallel Sorting Schemes. *IEEE Trans. on Computers*, 100(3):254–264, March 1983.

[85] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. Oracle TimesTen: An In-Memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[86] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H. Peter Hofstee, Gi-Joon Nam, Mark R. Nutter, and Damir Jamsek. Extrav: Boosting graph processing near storage with a coherent accelerator. *Proc. VLDB Endow.*, 10(12):1706–1717, August 2017.

[87] N Leischner, V Osipov, and P Sanders. GPU sample sort. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, April 2010.

[88] X Liu and Y Deng. Fast radix: A scalable hardware accelerator for parallel radix sort. In *2014 12th International Conference on Frontiers of Information Technology*, pages 214–219, December 2014.

[89] Xiaoyu Ma, Dan Zhang, and Derek Chiou. FPGA-Accelerated transactional execution of graph workloads. In *Proc. of the 2017 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, FPGA '17, pages 227–236, NY, USA, 2017. ACM.

[90] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. on Knowledge and Data Engineering*, 14(4):709–730, July 2002.

[91] R Marcelino, H C Neto, and J M P Cardoso. Unbalanced FIFO sorting for FPGA-based systems. In *2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009)*, pages 431–434, December 2009.

[92] Rui Marcelino, Horácio Neto, and João M P Cardoso. Sorting units for FPGA-Based embedded systems. In *Distributed Embedded Systems: Design, Middleware and Resources*, pages 11–22. Springer US, 2008.

[93] Maxeller Technologies. www.maxeler.com, 2015.

[94] Krishnan Meiyyappan, Liuxi Yang, Jeremy Branscome, Michael Corwin, Ravi Krishnamurthy, Kapil Surlaker, James Shau, and Joseph I. Chamdani. Accessing data in a column store database based on hardware compatible indexing and replicated reordered columns. Patent: US 20090254516 A1, 2009.

[95] MemSQL. `https://www.memsql.com`, 2013.

[96] Abhishek Mitra, Marcos R. Vieira, Petko Bakalov, Vassilis J. Tsotras, and Walid A. Najjar. Boosting XML filtering through a scalable FPGA-based architecture. In *In Proc. of 4th Conf. on Innovative Data Systems Research (CIDR)*, CIDR'09, 2009.

[97] Panagiotis Mousouliotis, Stavros Zogas, Panagiotis Christakos, Georzios Keramidas, Nikos Petrellis, Christos Antonopoulos, and Nikolaos Voros. Exploiting vitis framework for accelerating sobel algorithm. In *2021 10th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–5. ieeexplore.ieee.org, June 2021.

[98] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires: A Query Compiler for FPGAs. *Proc. of the VLDB Endowment*, 2(1):229–240, August 2009.

[99] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on FPGAs. *VLDB J.*, 21(1):1–23, February 2012.

[100] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-efficient aggregation: Hashing is sorting. In *Proc. of the 2015 ACM SIGMOD Int. Conf. on Management of Data*, SIGMOD '15, pages 1123–1136, NY, USA, 2015. ACM.

[101] Oracle Exadata. http://www.oracle.com/engineered-systems/exadata/index.html, 2014.

[102] M Owaida, D Sidler, K Kara, and G Alonso. Centaur: A framework for hybrid CPU-FPGA databases. In *2017 IEEE 25th Ann. Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218, April 2017.

[103] P Papaphilippou, C Brooks, and W Luk. FLiMS: Fast lightweight merge sorter. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 78–85, December 2018.

[104] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, and Others. Self-Driving database management systems. In *CIDR*, volume 4, page 1. cc.gatech.edu, 2017.

[105] Pico Computing. www.picocomputing.com, 2015.

[106] Constantin Pohl, Kai-Uwe Sattler, and Goetz Graefe. Joins on high-bandwidth memory: a new level in the memory hierarchy. *The VLDB Journal*, Jul 2019.

[107] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st Int. Symp. on*, ISCA'14, pages 13–24, 2014.

[108] Sanguthevar Rajasekaran and John H Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, June 1989.

[109] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. of the VLDB Endowment*, 6(11):1080–1091, August 2013.

[110] Suzanne Rivoire, Mehul A Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. JouleSort: A balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 365–376, New York, NY, USA, 2007. ACM.

[111] Bashar Romanous, Skyler Windh, Ildar Absalyamov, Prerna Budhkar, Robert Halstead, Walid Najjar, and Vassilis Tsotras. Efficient local locking for massively multithreaded in-memory hash-based operators. *VLDB J.*, 30(3):333–359, May 2021.

[112] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A Jacobsen. Multi-query Stream Processing on FPGAs. In *Proc. of the 2012 IEEE Int. Conf. on Data Engineering*, ICDE'12, pages 1229–1232, April 2012.

[113] Mohammad Sadoghi, Rija Javed, Naif Tarafdar, Harsh Singh, Rohan Palaniappan, and Hans-Arno Jacobsen. Multi-query stream processing on FPGAs. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1229–1232. ieeexplore.ieee.org, April 2012.

[114] Mohammad Sadoghi, Harsh Singh, and Hans-Arno Jacobsen. Towards Highly Parallel Event Processing Through Reconfigurable Hardware. In *Proc. of the Seventh Int. Workshop on Data Management on New Hardware*, DaMoN '11, pages 27–32, NY, USA, 2011. ACM.

[115] M Saitoh and K Kise. Very massive hardware merge sorter. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 86–93, December 2018.

[116] Nikita Shamgunov. The MemSQL In-Memory database system. In *IMDM@ VLDB*, 2014.

[117] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. of the 20th Int. Conf. on Very Large Data Bases*, pages 510–521, 1994.

[118] David Sheffield. IvyTown xeon+ FPGA: The HARP program. In *International Symposium on Computer Architecture (ISCA): Tutorial*, 2016.

[119] David Sidler, Zsolt Istvan, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. doppioDB: A hardware accelerated database. In *Proc. of the 2017 ACM Int. Conf. on Management of Data*, SIGMOD '17, pages 1659–1662, NY, USA, 2017. ACM.

[120] Valery Sklyarov and Iouliia Skliarova. High-performance implementation of regular and easily scalable sorting networks on an FPGA. *Microprocess. Microsyst.*, 38(5):470–484, July 2014.

[121] W Song, D Koch, M Luján, and J Garside. Parallel hardware merge sorter. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 95–102, May 2016.

[122] A Srivastava, R Chen, V K Prasanna, and C Chelmis. A hybrid design for high performance large-scale sorting on FPGA. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, December 2015.

[123] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database Analytics Acceleration Using FPGAs. In *Proc. of the 21st Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 411–420, 2012.

[124] N Tabrizi and N Bagherzadeh. An ASIC design of a novel pipelined and parallel sorting accelerator for a multiprocessor-on-a-chip. In *2005 6th International Conference on ASIC*, volume 1, pages 46–49, October 2005.

[125] Jens Teubner and Rene Mueller. How Soccer Players Would Do Stream Joins. In *Proc. of the 2011 ACM SIGMOD Int. Conf. on Management of Data*, SIGMOD'11, pages 625–636, 2011.

[126] Texas Advanced Computing Center. https://www.tacc.utexas.edu/, 2014.

[127] M. R. Thistle and B. J. Smith. A processor architecture for Horizon. In *Proc. of the 1988 ACM/IEEE Conf. on Supercomputing*, pages 35–41, 1988.

[128] Pınar Tözün, Brian Gold, and Anastasia Ailamaki. Oltp in wonderland: Where do cache misses come from in major oltp components? In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, New York, NY, USA, 2013. Association for Computing Machinery.

[129] Ali Vahidsafa, Sebastian Turullols, David Smentek, Ram Sivaramakrishnan, Paul Loewenstein, Sumti Jairath, and John Feehrer. The Oracle Sparc T5 16-Core Processor Scales to Eight Sockets. *IEEE Micro*, 33:48–57, 2013.

[130] K Vasanth, S Nirmal Raj, S Karthik, and P Preetha Mol. Fpga implementation of optimized sorting network algorithm for median filters. In *INTERACT-2010*, pages 224–229. ieeexplore.ieee.org, December 2010.

[131] Li Wang, Minqi Zhou, Zhenjie Zhang, Ming-Chien Shan, and Aoying Zhou. NUMA-Aware Scalable and Efficient In-Memory Aggregation on Large Domains. *Knowledge and Data Engineering, IEEE Trans. on*, 27(4):1071–1084, April 2015.

[132] Z Wang, H Huang, J Zhang, and G Alonso. Shuhai: Benchmarking high bandwidth memory on FPGAS. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119, May 2020.

[133] Skyler Windh, Prerna Budhkar, and Walid A. Najjar. CAMs As Synchronizing Caches for Multithreaded Irregular Applications on FPGAs. In *Proc. of the IEEE/ACM Int. Conf. on Computer-Aided Design*, ICCAD '15, pages 331–336, Piscataway, NJ, USA, 2015. IEEE.

[134] Skyler Arron Windh. *Hashing, Caching, and Synchronization: Memory Techniques for Latency Masking Multithreaded Applications*. PhD thesis, University of California, Riverside, 2018.

[135] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Da. Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance. *Xilinx Whitepaper*, 2017.

[136] Lisa Wu, Raymond J Barker, Martha A Kim, and Kenneth A Ross. Navigating big data with high-throughput, energy-efficient data partitioning. *SIGARCH Comput. Archit. News*, 41(3):249–260, June 2013.

[137] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *Proc. of the 19th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 255–268, 2014.

[138] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. Q100: the architecture and design of a database processing unit. *SIGARCH Comput. Archit. News*, 42(1):255–268, February 2014.

[139] Xilinx. Internals of bitonic sort. `https://xilinx.github.io/Vitis_Libraries/database/2020.2/guide/sort/bitonic_sort.html`. Accessed: 2021-6-17.

[140] Xilinx. Internals of insert sort. `https://xilinx.github.io/Vitis_Libraries/database/2020.2/guide/sort/insert_sort.html`. Accessed: 2021-6-17.

[141] Xilinx. Internals of merge sort. `https://xilinx.github.io/Vitis_Libraries/database/2020.2/guide/sort/merge_sort.html`. Accessed: 2021-6-17.

[142] Xilinx. Primitive APIs in xf::database: compoundsort. `https://xilinx.github.io/Vitis_Libraries/database/2020.2/guide/hw_api.html`. Accessed: 2021-6-17.

[143] Xilinx. Vitis libraries v2020.2. `https://xilinx.github.io/Vitis_Libraries/`. Accessed: 2021-6-17.

[144] Xilinx. *UG1314: Alveo U280 Data Center Accelerator Card*. Xilinx, February 2020.

[145] Xilinx. *AXI high bandwidth memory controller v1.0*. Xilinx, January 2021.

[146] Xilinx. *UG1120: Alveo Data Center Accelerator Card Platforms*. Xilinx, April 2021.

[147] Xilinx. *UG1416: Vitis Application Acceleration Development Flow Documentation*. Xilinx, March 2021.

[148] Xilinx. *UG1416: Vitis HLS*. Xilinx, March 2021.

[149] Xilinx Zynq. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, 2015.

[150] X Ye, D Fan, W Lin, N Yuan, and P Ienne. High performance comparison-based sorting algorithm on many-core GPUs. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, April 2010.

[151] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. Scalable Aggregation on Multicore Processors. In *Proc. of the Seventh Int. Workshop on Data Management on New Hardware*, DaMoN '11, pages 1–9, 2011.

[152] P. Yiannacouras and J. Rose. A parameterized automatic cache generator for FPGAs. In *Proc. IEEE Int. Conf. on Field-Programmable Technology (FPT)*, pages 324–327, December 2003.

[153] Marco Zagha and Guy E Blelloch. Radix sort for vector multiprocessors. In *Conference on High Performance Networking and Computing: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, volume 18, pages 712–721. Citeseer, 1991.

[154] C Zhang, R Chen, and V Prasanna. High throughput large scale sorting on a CPU-FPGA heterogeneous platform. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 148–155, May 2016.