University of California
Santa Barbara

Extracting Reusable Primitives of Key-Value Operations and Efficient Architecture Support

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

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Bangyan Wang

Committee in charge:

Prof. Yuan Xie, Chair
Prof. Zheng Zhang
Prof. Yufei Ding
Prof. Kerem Camsari

September 2023

The Dissertation of Bangyan Wang is approved.

Prof. Zheng Zhang

Prof. Yufei Ding

Prof. Kerem Camsari

Prof. Yuan Xie, Committee Chair

June 2023

Extracting Reusable Primitives of Key-Value Operations and Efficient Architecture Support

by

Bangyan Wang

Acknowledgements

I thank everyone who helped me in this journey.

First, I would like to express my heartfelt gratitude to my advisors, Prof. Yuan Xie and Co-Advisor Prof. Zheng Zhang, for their unwavering guidance, support, and encouragement throughout my Ph.D. study. Their invaluable insights and expertise have been instrumental in shaping my research and academic growth. I am truly grateful for their mentorship, and I will always cherish the lessons I learned from them.

I want to extend my appreciation to my family for their unconditional love and unwavering support. Their constant encouragement and motivation helped me stay focused during challenging times, and I am deeply grateful for their presence in my life.

I am also grateful to my collaborators, including the SEAL-members Liu Liu, Lei Deng, Zheng Qu, Shuangcheng Li, Zhaodong Chen, and the professors and students from Tsinghua, Guohao Dai, Tianyu Fu, Chiyue Wei, Xiangyu Li, Huazhong Yang, and Yu Wang. Their contributions to my research have been invaluable, and I am honored to have had the opportunity to work with such talented and dedicated individuals.

Furthermore, I would like to express my gratitude to my mentors and colleagues at Alibaba DAMO Academy and Amazon, Fei Sun and Sheng Xu, for their guidance, support, and inspiration during my three internship programs. Their mentorship provided me with invaluable industry experience and helped me gain a broader perspective on my research.

Finally, I would like to thank everyone who supported me during my Ph.D. journey, including my friends, colleagues, and all those who have been a part of my academic journey. Your support and encouragement have been truly appreciated, and I could not have achieved this milestone without you.

Curriculum Vitæ
Bangyan Wang

Education

2017-	Ph.D. ECE, University of California, Santa Barbara.
2016 Feb∼Aug	Exchange Student, ETHz, Zurich, Switzerland.
2013-2017	B.S in Electronic Engineering, Tsinghua University, Beijing, China.

Publications

[1]. Bangyan Wang, Guohao Dai, Yuan Xie "A One-for-all Architecture for the Reduce-by-Key Operation: A Coordinated Parallelism, IO, and Storage Perspective" (In submission)
[2]. Bangyan Wang, Lei Deng, Fei Sun, Guohao Dai, Liu Liu, Yu Wang, Yuan Xie "A One-for-All and $O(V\log(V))$-cost Solution for Parallel Merge Style Operations on Sorted Key-Value Arrays" Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2022
[3]. Guohao Dai, Zhenhua Zhu, Tianyu Fu, Chiyue Wei, Bangyan Wang, Xiangyu Li, Yuan Xie, Huazhong Yang, Yu Wang "DIMMining: Pruning-Efficient and Parallel Graph Mining on Near-Memory-Computing" 49th IEEE/ACM International Symposium on Computer Architecture (ISCA), 2022
[4]. Bangyan Wang, Lei Deng, Zheng Qu, Shuangcheng Li, Zheng Zhang, Yuan Xie "Efficient Processing of Sparse Tensor Decomposition via Unified Abstraction and PE-interactive Architecture" IEEE Transactions on Computers, 2021
[5]. Zhaodong Chen, Lei Deng, Bangyan Wang, Guoqi Li, Yuan Xie "A Comprehensive and Modularized Statistical Framework for Gradient Norm Equality in Deep Neural Networks" IEEE Transactions on Pattern Analysis and Machine Intelligence,2020
[6]. Zheng Qu, Bangyan Wang, Lei Deng, Hengnu Chen, Jilan Lin, Ling Liang, Guoqi Li, Zheng Zhang, Yuan Xie "Hardware-Enabled Efficient Data Processing with Tensor-Train Decomposition" IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,2021
[7]. Hongzhi Zuo, Manxiu Cui, Bangyan Wang, Cheng Ma "A spectral-domain model-based method for simultaneous oxygen saturation quantification and contrast agent identification" Optics in Health Care and Biomedical Optics XI,2021
[8]. Manxiu Cui, Hongzhi Zuo, Bangyan Wang, Xuanhao Wang, Cheng Ma "A convex cone method for accurate blood oxygenation photoacoustic imaging" Proceedings Volume 11550, Optoelectronic Imaging and Multimedia Technology,2020

Abstract

Extracting Reusable Primitives of Key-Value Operations and Efficient Architecture Support

by

Bangyan Wang

The advancement of general-purpose architecture has reached a juncture where the continuing investment to improve instruction per cycle (IPC) yields diminishing returns. While domain-specific architecture more efficiently converts silicon resources into throughput, economic viability hinders their wider adoption, except in a few areas. A more feasible way is to extract reusable operations that can be used across multiple domains and then find efficient architecture to support them.

This dissertation focuses on operations involving pairs of keys and values. The application spans a wide range of domains, including database, graph computing, genomics, and sparse computing. The processing of key-value pairs is divided into two categories: ordered and unordered. For the ordered category, we optimized the general merge style operations on a sorted key-value array by creating a set of highly composable primitives. Next, we show that many widely used ordered data structures and algorithms, such as heap and binary search, can be accelerated by rewriting them to use merge operation as a building block. For the unordered ones, we observe that reduce-by-key is a common bottleneck in many domains. We propose the design of the Reduce-By-Key core and introduce a new algorithm to accelerate this operation. We also analytically prove that our method is close to optimal. Lastly, we investigate the decomposition operation on sparse tensors - a special form of key-value pairs. We show how a PE-interactive architecture can be used to significantly increase data reuse.

Contents

Chapter 1

Introduction

In the past decades, the single-core performance of general-purpose processors has been improved by only ~ 2, suggesting the point of diminishing returns has been reached for the classical path of improving instruction-per-cycle(IPC). The increase in the core count has been the main driver of performance improvement for both CPU and GPU, but the slowdown of Moore's law, in particular, the stall transistor-per-dollar ratio, has made it less efficient to continue in this direction. Seeing scheduling and decoding of instructions as an expensive tax added to the useful computation, a trend in recent years is to hardcode more complex computation directly in hardware to amortize this overhead. Both domain-specific accelerators (DSA) and domain-specific processors (with customized instruction set) are representative examples of this approach. However, the cost of developing a new DSA or processor is high, and the economic viability prevents their wider adoption except in a few areas with huge market demands, such as AI.

To optimize the cost-benefit ratio of specializing hardware, it is essential to extract reusable operations that can be applied to multiple domains. This thesis focuses on the process of (key,value) pairs in associative containers. Those (key,value)-related operations are often used as workhorses across multiple domains, including AI, database, genomic

processing, graph analysis, and scientific computing. At the same time, (key,value)-related operations are challenging for modern processors with high parallelism (such as SIMD on CPU, and GPU) because they were designed to provide high computation and memory throughput for regular arrays, such as dense arrays or matrices. (key-value)-related operations demand equally efficient key-based search, comparison and maneuver capability, which is challenging in both computation and memory subsystems for conventional architecture.

1.1 Motivation

We divide the (key,value)-related operation into two categories and they are handled separately.

- Ordered: Certain ordered constraints exist between the keys, and those constraints must also be preserved in output after the operation. The keys must have defined the $\leq$ operator and $=$ operator.
- Unordered: No ordering relation is defined between the keys in the input, nor is it required in the output. The keys must have defined the $=$ operator.

The support for the (key,value) operations is motivated by the following applications:

Scientific computing: The processing of sparse vectors, matrices, and tensors is the central computation workload in many scientific computing scenarios, including circuit simulation (e.g. SPICE) and psychic simulation (e.g. finite element method), etc. Those sparse arrays are usually encoded directly as a sorted (key, value) array where the key is the index of the non-zero element. Due to the high sparsity, arithmetic operations over them, including addition ($+$), element-wise multiplication ($.*$), and multiplication ($\times$) should only iterate through the non-zero elements, and all scalar operations must be

performed on values with matched keys. Being sorted by keys allows the key-matching between those sparse arrays to be efficient, and preserving the ordered constraint is necessary for subsequent operations.

Database: The tables in a relational database are naturally (key,value) arrays. The table can be sorted or not, and many database operations can be seen as ordered/unordered (key,value) operations. This include 1) various join-operations are defined between two or more tables to produce a new table. 2) group-by aggregation requires de-duplicating the keys meanwhile aggregating the values with identical keys. 3) query or partition all tuples in a table into bins according to the range where their keys belong to. The range is usually defined by another sorted (key,value) array. 4) sorting, which transforms an unordered (key,value) array into an ordered one.

Graph: Graph is a typical data structure in many AI applications, such as recommendation systems, fraud detection, and social network analysis. The graph is usually represented as adjacency lists, a sorted (key,value) array where the keys are the ID of neighbor nodes, and the values are some edge properties. Two types of (key,value) operations are widely used in graph processing: 1) the set operations (union, intersection, difference, etc.) between two lists, which are used in analytic graph algorithms to extract graph patterns, such as clique-counting. 2) the push-update operation, where a small active set of nodes is pushing updates the union of all their neighbors through the edges, leading to an operation similar to the group-aggregation in databases.

Genomic Analysis: K-mer counting in a DNA/RNA sequence is the foundation step of many genomic analysis applications. It can be seen as reducing (key,value) tuples where the keys are the K-mer and the values are the counts of the K-mer in the sequence.

1.2 Challenge, Approach, and Contribution

We seek to design a minimized, multi-domain, performant, "confident" architecture for (key, value) operations, and such architecture can be easily integrated into general purposed processors or accelerators.

- Being minimized means both the extra circuit components and primitive definitions added to the architecture to be minimized. However, a unified encoding of (key, value) operations that can be used in multiple domains is not obvious. Avoiding inflation of the hardware size, ISA, and compilers requires novel ways of viewing and decomposing those operations.

- Being performant means the architecture should be highly parallel and memory-friendly. However, many (key, value) operations are inherently sequential (for ordered operations), write-conflict-heavy (for unordered operations), and have a bad memory access pattern (unordered operations).

- Being "confident" means we know how far we are from the optimal solution. This requires analysis and proof of the performance-critical criteria on both the proposed and optimal theoretical design. Those criteria include, for example, the memory footprint, traffic, and the "randomness" of random-memory access.

  It's common in architecture research to numerically evaluate a proposed design against a reference design and report the relative speedup. However, it was rarely known how much room was left for improvement.

With the above pursuits, our approach follows three major steps: 1) find a unified abstraction for the targeted problem domain, 2) design a performant architecture, and 3) provide a theoretical analysis of performance-critical criteria.

For ordered operations, the key observation is the unique role of one particular family (key, value) operations, which we named the merge style operation (MSO) on the sorted key-value array. MSO can efficiently implement other ordered operations with appropriate algorithm rewrite.

- Our first work tackles the MSO on CPU SIMD platform. To address the challenge of the diversity of MSO operations, we 1) proposed a unified abstraction called "restricted zip machine." to encode all possible MSO operations. 2) designed a set of highly composable SIMD primitives such that only four new instructions are required to implement arbitrary MSO operations. To address the challenge of the sequential dependency of MSO operations and the quadratic hardware cost ($O(V^2)$ for SIMD width V), we 3) designed a highly parallel hardware implementation with only $O(V(\log(V)))$ hardware cost.

- Our second work expands the usability of MSO SIMD primitives to even broader applications. We show that more sophisticated ordered data structures/algorithms can be rewritten based on MSO. They include binary heap, binary search (batched), and k-way additive merge.

For unordered operations, we focus on addressing the bad memory access pattern that make memory subsystem the performance bottleneck. We investigated two operations on unordered (key, value) pairs.

- Our third work look at the "reduce-by-key" operation. We propose to use an incremental de-duplication scheme on "sorted, deduplicated hash array (SDHA)" to finish this operation. Comparing to the existing methods, the advantage of this methods is it successfully addressed the profound conflicts between multiple optimization goals including parallelism, traffic reduction, and memory footprint and the memory access pattern.

In this work, we show 1) a new algorithm that are suitable in the context of parallel hardware that share a memory hierarchical, 2) the hardware design of "reduce-by-key" core to implement the algorithm, and 3) an analysis on the near-optimality of memory footprint, memory traffic, and randomness of random memory access.

- Our fourth work look at the decomposition of sparse tensors. Sparse tensors can be regard as a special form of (key, value) pairs. We designed an accelerator for sparse tensor decomposition with a new PE-to-PE communication protocal to help reduce the off-chip memory traffic and improve performance.

1.3 Related Work

The technique proposed in this thesis for efficiently processing key-value data has broad applications across domains, including scientific computing and graph analytics. Although research into computer architectures in these domains has been active for many years, the processing of key-value data represents a higher-level abstraction than the individual domain-specific operations. To my knowledge, no prior work has approached the subject from this perspective. Consequently, the related work section will introduce only the domain-specific hardware designs by subdomain, while detailed surveys of related work will appear in the corresponding chapters.

Both scientific computing and artificial intelligence (AI) frequently require processing large numerical tensors (multi-dimensional arrays). Matrix-matrix and matrix-vector multiplications are among the most common operations. The dense variants of these operations have been thoroughly studied and optimized over the past decades. For instance, Nvidia's GPUs with tensor cores and Google's TPUs with systolic arrays possess dedicated hardware units for matrix-matrix multiplication, achieving unparalleled energy efficiency (exceeding 2TFLOPs/Watt). However, the sparse versions of these operations

present more of a challenge due to data irregularities and the need for key-based searches and matches. Many hardware designs utilize a sorted-set intersection/union algorithm to identify matched keys from two sparse arrays. The primary hurdle is the sequential dependency of such operations, which hampers parallelization. Furthermore, existing designs typically cater to specific types of sparse tensor operations, such as sparse matrix-vector multiplication (SpMV) and sparse matrix-sparse matrix multiplication (SpGEMM).

Databases are commonly organized into tables, where columns function as keys. Many database operations, like join operations, sorting, and group-by aggregation, naturally align with key-value operations. Offloading database operations to FPGAs was a hot research topic over the last decade. However, industry adoption remains limited. The primary hindrance is that offloading proves more complicated than other acceleration methods requiring only software modifications, like utilizing SIMD instructions.

Similarly, genomics and graph analysis involve data structures interpretable as key-value pairs. Despite numerous accelerator proposals over the past decades, the scenario mirrors database acceleration: offloading to standalone hardware complicates the software stack, and the performance gains don't sufficiently justify the associated costs.

Chapter 2

Merge Style Operation on Ordered Data

In this chapter, we investigate merge-style operations (MSO) on sorted key-value arrays. MSO is widely used in scientific computing, deep learning, databases, graph analysis, sorting, set operations, and more. MSOs dominate the execution time in important applications such as SpGEMM (by 90%) and graph mining. However, MSO is a SIMD-unfriendly operation. This is not only because MSO cannot be efficiently implemented using existing SIMD instructions available in ISAs that are widely used commercially (e.g., x86, ARM, RISC-V), but also because even defining a set of feasible SIMD instructions that can implement MSO is a challenging task. By "feasible", we mean achieving performance gains, maintaining low hardware costs, minimizing ISA modifications, and providing sufficient flexibility to cover all relevant use cases. In this chapter, we present a solution to this problem.

2.1 Background

We provide a brief background on merge style operations (MSOs) in this section, including their definitions, applications, existing solutions, and the limitations of these

solutions.

2.1.1 Definition

MSOs are very similar to merge sort but are more general. They take two sorted key-value arrays as inputs and produce another sorted array as output, as illustrated in Figure 2.1(a). They utilize a binary function $op(x, y)$ to process pairs of values with matched keys. Additionally, they might remove, insert, or duplicate certain elements when dealing with matched or mismatched keys based on specific rules referred to as "patterns". These patterns include OR, AND, SORT, and Range-Match patterns, as illustrated in Figure 2.1(b)-(e).

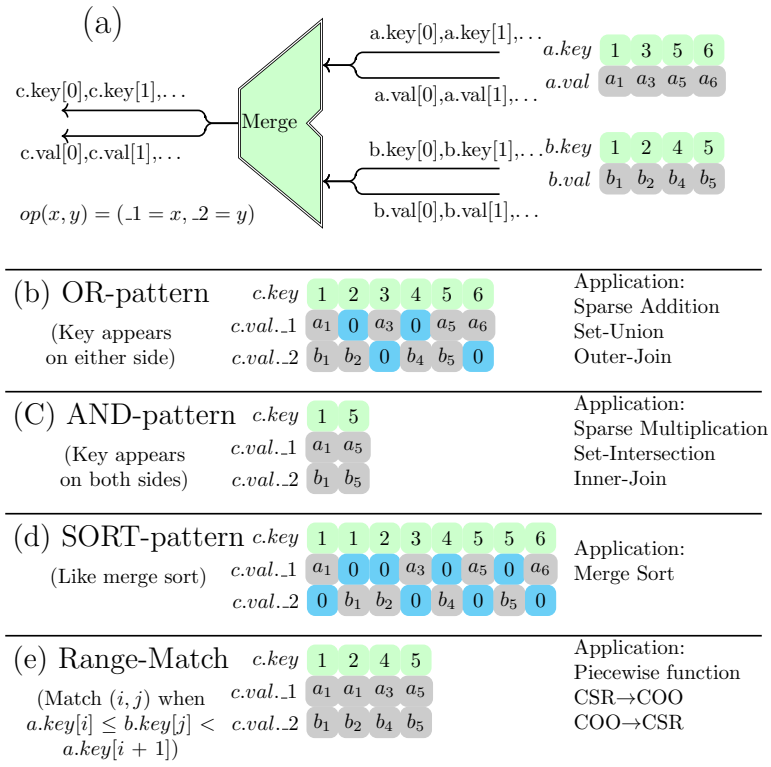


Figure 2.1: MSOs and the patterns.

2.1.2 Applications

Scientific computing and deep learning. Many scientific computing and deep learning problems involve sparse vectors/matrices/tensors. The coordinate (COO) format can be seen as a key-value array, while the compressed sparse row (CSR) format can be viewed as a collection of key-value arrays wherein each one represents a sparse row. Several types of operations on sparse vectors/matrices/tensors can be implemented as MSOs. (1) Element-wise sparse addition and multiplication can be implemented with OR-pattern and AND-pattern. (2) Sparse matrix multiplication can be implemented as weighted sum of sparse vectors, which in turn becomes OR-pattern. (3) The conversion between COO↔CSR formats requires a kind of transform between arrays like [0,0,0,1,1,2,2,2]↔[0,3,5,8] which can be implemented using Range-Match-pattern. Another application of Range-Match-pattern is numerical interpolation for piecewise functions. (4) The transpose of sparse matrices/tensors is partially a sort problem, which is SORT-pattern.

Graph analytics. Many algorithms in graph analytics involve operations similar to sparse matrix multiplication, but with + and × operators replaced by other operators, and the value type is usually a data structure instead of a 32-bit floating-point number. These algorithms can be implemented using the OR-pattern. Additionally, graph mining can be implemented using set intersection (AND-pattern) and set difference (Diff-Pattern).

Sorting. Merge sort is the most commonly used stable sorting algorithm. Its basic building block involves merging two sorted key-value arrays into one longer key-value array, which is a SORT-pattern Merge-Sort Operation (MSO).

Databases. Join operations combine information from two tables into one table based on matching keys. There are multiple variations of join operations, as depicted in Figure 2.2. They can be implemented as MSOs using different patterns.

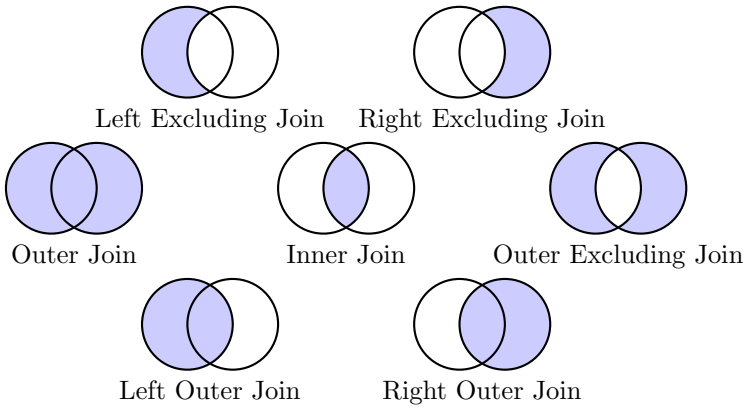
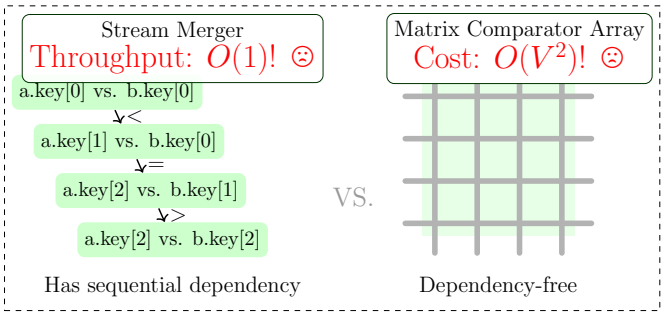


Figure 2.2: Various forms of join operations in database.

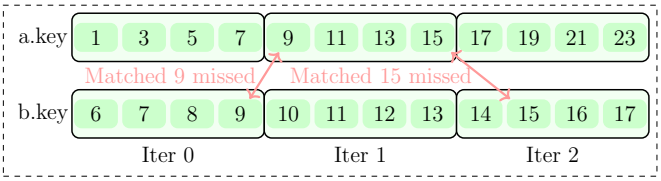
2.2 Challenges and Related Work

Sequential design results in low throughput, while parallel design incurs high costs. An MSO can be implemented either sequentially or in parallel. Sequential design can be accomplished through a FIFO merger that takes two FIFOs as input ports. In each cycle, the heads of the two FIFOs are compared, and the smaller one is popped. The advantages of sequential design include simplicity and low cost. Additionally, this FIFO merger can easily accommodate various merge patterns, such as the AND-pattern [1, 2, 3] and OR-pattern [4, 5, 6]. However, its main limitation is the 1-tuple/cycle throughput, resulting from the sequential dependency in comparing and popping keys from the FIFOs (as shown on the left in Figure 2.3-Challenge 1).

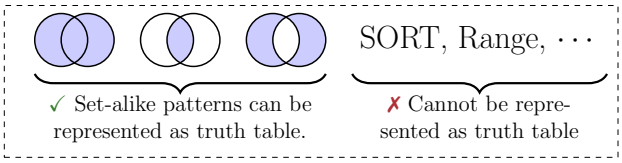
To eliminate the dependency of later comparisons on previous ones, parallel design often involves performing all-to-all key comparisons within a local scope of V elements. This approach yields V^2-bit comparisons, leading to higher resource costs, as illustrated



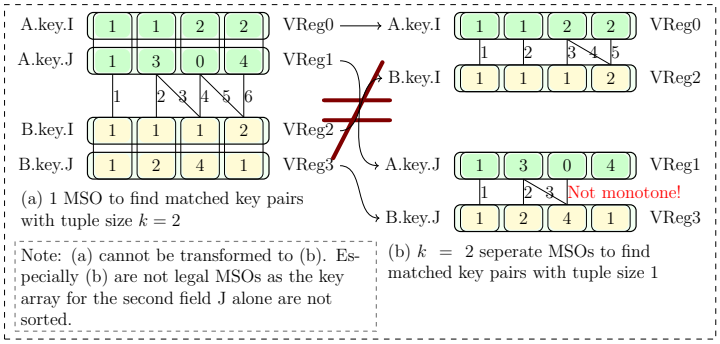
Challenge 1: Sequential design suffers low throughput, while parallel design suffers high cost



Challenge 2: Supporting long input arrays



Challenge 3: Lacking uniform representation of matching patterns



Challenge 4: Supporting variable tuple size via decomposition

Figure 2.3: Challenges of MSOs.

Table 2.1: Latency of x86 all-to-all comparison instructions [14].

Instruction	V	Latency
PCMPESTR	4	19 cycles
VPCONFLICT_256	8	37 cycles
VPCONFLICT_512	16	67 cycles

Table 2.2: Supported patterns using known techniques.

Pattern	Accelerators		CPU	
	Sequential	Parallel	Sequential	Parallel
AND-Pattern	✓	✓	✓	✓
SORT-Pattern	✓	✓	✓	✓
OR-Pattern	✓	✓	✓	✗
Other Patterns...	✓	✗	✓	✗

on the right in Figure 2.3-Challenge 1. Accelerators typically face a trade-off between the $O(V^2)$ cost and the $O(1)$ performance [7, 8, 9]. On SIMD-based CPUs, some studies [10] utilize specialized x86 instructions for all-to-all comparison, but these instructions suffer from long latency (Table-2.1). Other works employ last-byte-check and shuffle techniques to achieve an equivalent $\sqrt{V} \times \sqrt{V}$ comparison [11, 12]. After obtaining the V^2-bit result, key matching is generated using specialized circuits (in accelerators) or pre-computed lookup tables. For intersection (AND-pattern), the table requires 2^V entries [10]. However, for other patterns (e.g., OR-pattern), the table would necessitate an impractical 2^{V^2} entries, making it inapplicable. In the case of merge sort, both accelerators and SIMD implementations benefit from the sorting network technique introduced by Ken Batcher in 1968 [13], which requires only $O(V \log(V))$ comparisons. Nonetheless, this method is only suitable for SORT-pattern operations. Currently known supported patterns using sequential or parallel design are summarized in Table 2.2.

Supporting long input arrays. In hardware, the width V for parallel execution (e.g., SIMD width on a CPU) is often fixed (e.g., 4, 8, or 16), whereas the two input key-value arrays can have arbitrary lengths. To address this challenge, loop-tiling can be employed

to divide the original problem into multiple sub-problems with shorter lengths. However, applying loop-tiling to MSOs is more intricate compared to its element-wise counterpart and does not yet have a universal solution.

Primarily, the naive approach to loop tiling is not viable for nearly all MSOs:

```
Merge(a[0:end],b[0:end]) != Concat(
    Merge(a[0:V],b[0:V]),
    Merge(a[V:2V],b[V:2V]),
    Merge(a[2V:3V],b[2V:3V])...)
```

Figure 2.3-Challenge 2 provides an example of how matched elements can be missed (9 and 15) if we naively divide the 12-element input arrays into 3 sub-problems ([0:4], [4:8], [8:12]). Partial solutions do exist for the AND-pattern and SORT-pattern. For the AND-pattern, a commonly employed approach involves comparing the last element of the two input vectors and advancing the smaller side [10, 7, 11, 12, 8]. In the case of the SORT-pattern, one method is to retain the larger half (V elements) of the sorted result ($2V$ elements) for the next iteration and sort using the V elements from the smaller side of the two input arrays [15, 7]. However, these solutions are conditionally correct (as they do not support inputs with duplicated keys), inefficient in terms of work (resulting in a waste of 50% of useful operations), and most importantly, they cannot be generalized to other MSO patterns, such as the OR-pattern. Unfortunately, no comprehensive solutions have been identified in the literature that apply universally across all known matching patterns.

The lack of uniform representation of matching patterns. There are no strict models that properly encode different patterns and handle them uniformly (see Figure 2.3-Challenge 3). A truth table is a good candidate to cover set-operation-related patterns (e.g., those in Figure 2.2), but it cannot accommodate SORT, Range-Match, or other one-to-many-

match patterns. Most prior work focuses on only one or two specific patterns and does not address this problem.

Support variable tuple sizes via decomposition. Many applications of MSOs require the key and value to be tuples comprising multiple 32-bit fields. Given the fixed nature of hardware, a single SIMD operation cannot directly support variable key tuple sizes. The only feasible approach is to decompose an MSO, whose key tuple contains k 32-bit fields, into k fixed-sized operations. The challenge, however, is that one MSO with a key tuple size of k vastly differs from k independent MSOs each having a key tuple size of one. Moreover, each of these MSOs processes just one field of the original problem, as depicted in Figure 2.3-Challenge 4. This difference obstructs the previously mentioned decomposition approach. This same challenge extends to programmable platforms like CPU/GPU, not just to accelerators. The scenario here contrasts sharply with element-wise operations, where complex element-wise operations can be easily broken down into a finite set of basic element-wise operations such as $+$, $-$, $\times$, $<$, etc. As of now, no prior work support variable key sizes, with the exception of a CGRA-based design [3], which can modify the hardware datapath during runtime.

Discussion: Bandwidth and data reuse rate. A single MSO operation, in isolation, has relatively low arithmetic intensity. This suggests that, with advancements in the computation aspect of MSO, there's a need to either promote higher-level data reuse or provide increased memory bandwidth. When applications use MSO as a foundational building block, there is typically significant potential for data reuse across multiple MSOs. Examples include employing a merge tree buffered on-chip to implement multi-way merges [15, 9] or leveraging application-specific data reuse techniques [8]. As an alternative, accelerators often contemplate using emerging technologies, such as HBM, to furnish

greater memory bandwidth. Although this paper concentrates on the computational aspect of MSO, capitalizing on higher-level data reuse for each unique application can be achieved at the software layer and isn't the central focus of this work.

2.3 Uniform Representation

This section addresses the challenges of finding a uniform representation to encode matching patterns by introducing two sequential models: the general zip machine (GZM) and the restricted zip machine (RZM). The GZM is a more general yet strictly sequential machine. After partially specializing the state machine within the GZM, we derive the RZM. Any RZM can be implemented using the proposed SIMD primitives, which are detailed in Section 2.4.

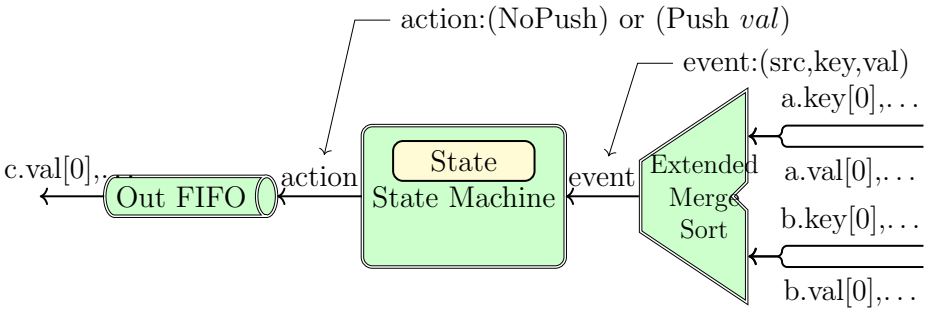


Figure 2.4: General zip machine (GZM).

2.3.1 General Zip Machine (GZM)

The MSO represented in Figure 2.1(a) can be reinterpreted as "a 2-to-1 merge sort followed by a state machine," as depicted in Figure 2.4. The underlying rationale for this transformation is that identical keys become neighbors after a 2-to-1 merge sort, simplifying key matching. Consequently, the primary stage of the GZM comprises an "extended merge sort unit" that sorts two key-value arrays. In addition, it logs the

origin of each output key-value pair. The output is a sequence of tuples (src, key, val), where src can be 'a' or 'b' (1-bit); we refer to each tuple as an event. These events are then channeled to a state machine that processes them sequentially. Typically, "state" is utilized to retain a sliding window of the event stream, enabling the discovery of matching keys within this window. For each incoming event, the state machine produces an action. This action can either instruct the "Out FIFO" to append a value val at the end (PushResult val) or do nothing (NoPush). Any MSO can be represented as a GZM by selecting a suitable state machine. For instance, "sparse vector multiplication" can utilize a state machine that maintains a sliding window of 2 events; if their keys align, their val segments are multiplied ($op(x, y) = x \times y$), and the result is pushed. If there's a mismatch in keys, nothing is pushed (NoPush). Other MSOs, such as "sparse vector addition" and "set-diff", have their unique state machines. For one-to-one matching, the state typically encompasses a sliding window of 2 or 3 events. In one-to-many matching, a key might replicate indefinitely, seemingly requiring an infinitely long sliding window. However, this can be circumvented by embedding supplementary data within the state while retaining a finite sliding window in the state machine.

It's noteworthy that the key ($c.key[0], \ldots$) is absent from the output array in Figure 2.4 (only $c.val[0], \ldots$ is visible) compared to Figure 2.1(a). This omission retains generality: we can always retrieve the key by constructing an auxiliary GZM wherein the original key constitutes part of the proxy value. For example, one can assign $ValC' = (Key, ValC)$, $ValA' = (Key, ValA)$, $ValB' = (Key, ValB)$, $op'((kx, vx), (ky, vy)) = ((kx|ky), op(vx, vy))$ in the proxy GZM to concurrently extract Key and $ValC$ from $ValC'$.

2.3.2 Restricted Zip Machine (RZM)

The RZM is derived from the GZM by constraining the state machine in the GZM to a fixed pipeline, with only a few components being configurable. Nevertheless, the RZM remains potent enough to encompass intriguing use cases such as OR-pattern, SORT-pattern, and so on. We will first discuss several foundational ideas prior to delving into its formal definition.

Constructing two operand streams: The state machine in the GZM transforms a stream (the event stream) into an output stream (the $c.val[0], \ldots$). This process can consistently be divided into two phases. 1) Construct two operand streams from the event stream, which correspond to the two operands of the binary operator $op(x, y)$. 2) Implement the operator $op(x, y)$ element-wise to the two operand streams, generating the desired output stream. Figure-2.5 illustrates an instance of the set-union operation for the inputs $a = [1, 2, 5, 8]$ and $b = [1, 3, 5, 7]$. In this scenario, the two operand streams have all matching elements perfectly aligned, with zeroes judiciously placed where necessary. Subsequently, applying an element-wise bitwise-OR to the two streams yields the union of the sets $c.val = [1, 2, 3, 5, 7, 8]$. (Given that the key and value for each event in the event stream are identical in this example, they are portrayed on a singular row to conserve space in Figure-2.5 and subsequent illustrations. Blue zeroes are inserted while the rest are derived from the value portion of the event stream.)

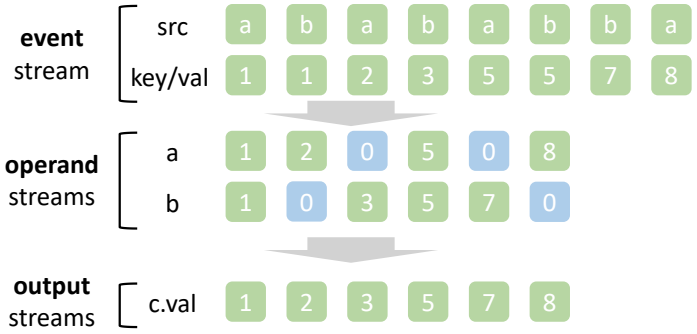


Figure 2.5: Operand streams as an intermediate step

Employing FIFO and commands to construct operand streams: A single operand stream can be formed using a FIFO, which accepts a command stream associated with the event stream. We have identified four distinct command types, as delineated in Table-2.3. By selecting the appropriate commands, data can be aligned in various manners to match the desired pattern. Figure-2.6 provides an illustration, crafting the operand streams from the aforementioned set-union example. Additionally, Figure-2.7 offers another instance, showcasing how the PushMostRecent command can facilitate the implementation of one-to-many matching patterns.

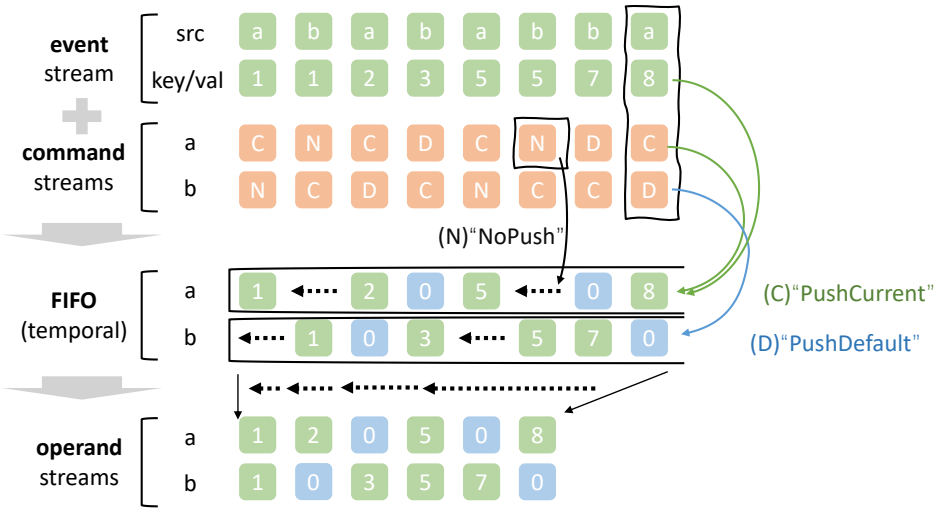


Figure 2.6: Use FIFO and command streams to build operand streams

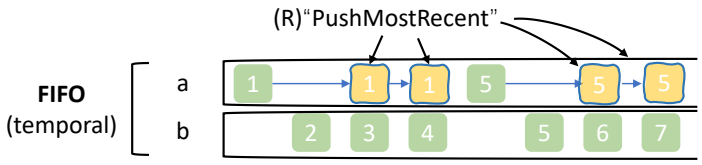


Figure 2.7: Extra example: Use PushMostRecent to implement one-to-many range-match

Table 2.3: Behaviors of M-FIFOs.

actions.a/b	Effect
(N)NoPush	Do nothing
(C)PushCurrent	Push the value of current event
(R)PushMostRecent	Duplicate the last element in the FIFO
(D)PushDefault	Push default value (usually 0)

Commands can be deduced from a sliding window of three events: The command streams are derived from the event stream. For each event, its corresponding commands can be determined by only considering its immediate neighbors (left and right) along with the event itself. This is rooted in the fact that commands are contingent upon the presence of a matched pair for the given event. After the merge sort, a check on the two neighbors suffices to ascertain the existence of such a pair.

Every matching pattern can be encoded as a function mapping a 5-bit input to a 4-bit output: Within the sliding window of three events, namely event[i-1], event[i], and event[i+1], each event takes the form of a tuple (src, key, val). However, only 5 bits of information are crucial, as depicted in Figure-2.8. The val component and the absolute magnitude of key don't influence the matching determination. What truly counts are the src values of the three events and the comparative relationships $(<, =, >)$ of the keys. Additionally, given that keys are already arranged in a non-decreasing order within the event stream, only two points of comparison can yield different outcomes: the comparison of event[i-1].key and event[i].key, as well as that of event[i].key and event[i+1].key. Diverse matching patterns are differentiated by the two commands they

generate for the possible 2^5 scenarios, which means they can be represented as functions mapping 5-bit inputs to 4-bit outputs, or equivalently, 128-bit values ($2^5 \times (2+2)$ bits). In subsequent sections, we refer to the stream of these "5-bits" as the info stream. An exemplification of encoding the range-match pattern into such a 5-bit to 4-bit function, along with its application, is portrayed in Figure-2.9[1].

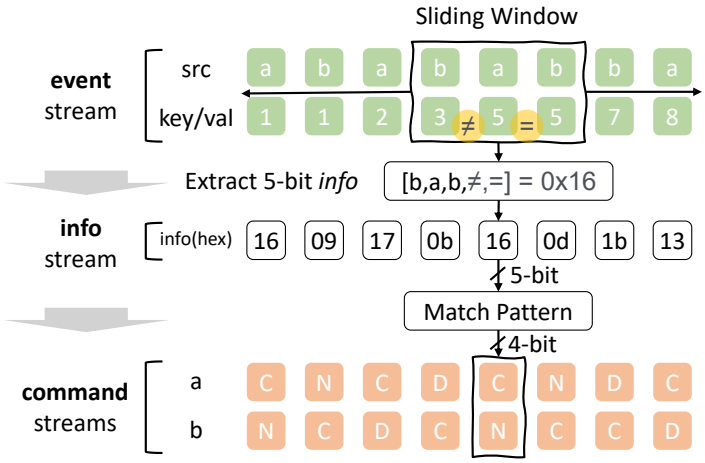


Figure 2.8: Extracting 5-bit info from the sliding window[pattern: union pattern (one-to-one)]

Put everything together: The RZM's architecture, as depicted in Figure-2.10, initiates by merging two input streams into the event stream, mirroring the functionality of GZM. Successively, by applying a function (which encodes the desired pattern) to the 5-bit section of each sliding window comprising three elements, two command streams (labeled as $actions.a/b$) are generated. Following this, with the aid of two FIFOs (denoted FIFO A/B), the event stream, and the two command streams, two operand streams are constituted. The terminal phase involves inspecting the pairs from both FIFOs: when both FIFOs contain elements, the operator $op(x, y)$ is applied to the fetched pair, and the outcome is appended to $c.val$. Otherwise, no action is taken.

[1]In this depiction, for simplification, we have retained identical values for key and input. However, in practical scenarios, values often deviate from keys.

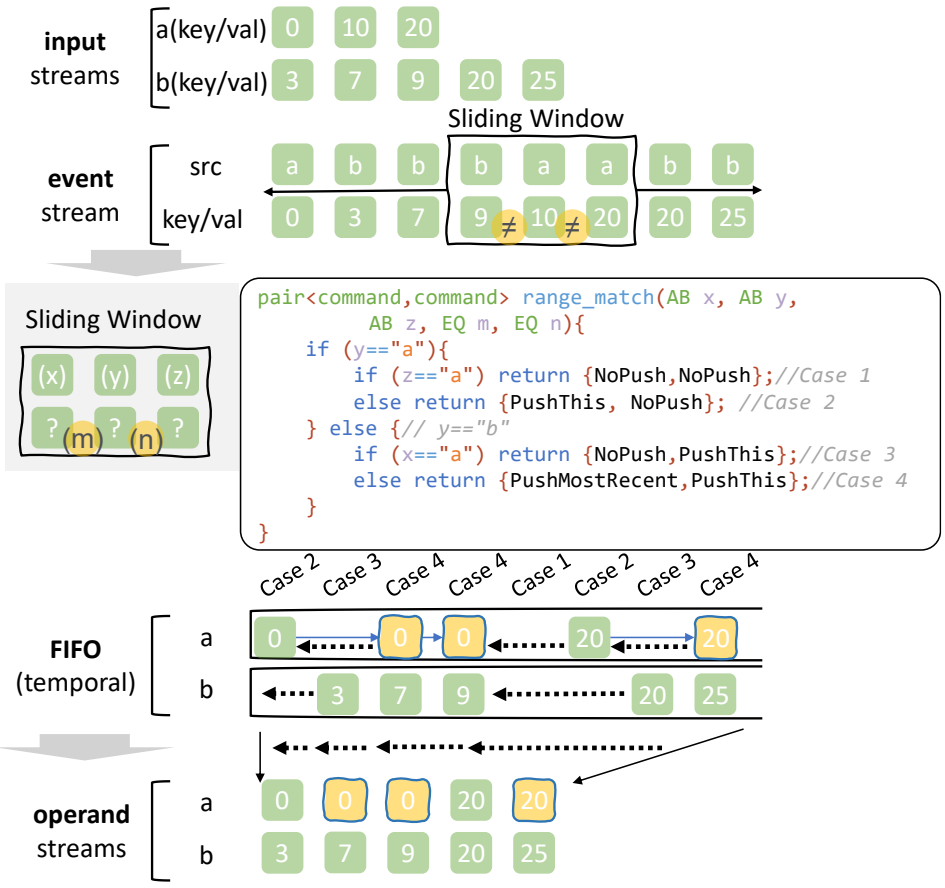


Figure 2.9: Example: Use range-match pattern to put b=[3,7,9,20,25] into bins [0,10),[10,20),[20,+∞) defined using delimiters a=[0,10,20]. The 5-bit are named as [x,y,z,m,n] respectively.
[pattern: range-match pattern (one-to-many)]

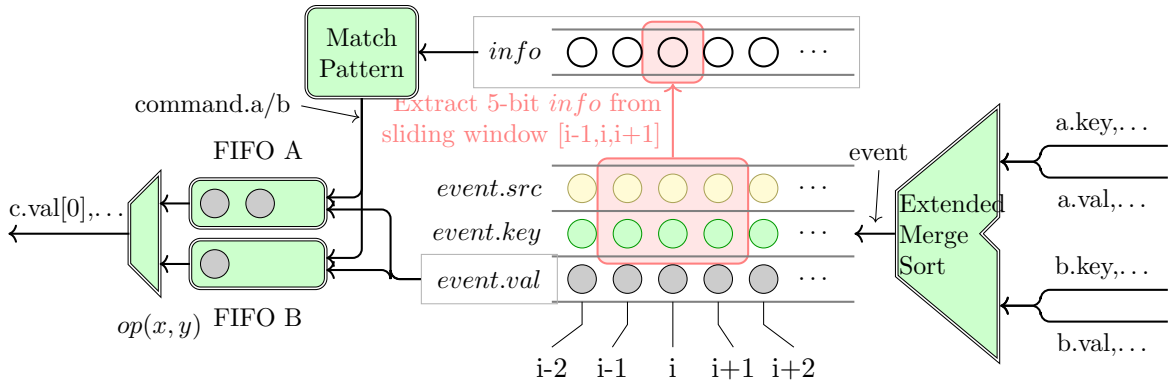


Figure 2.10: Illustration of the Restricted Zip Machine (RZM).

2.3.3 Exploring the Divide-and-Conquer Relation

In this section, we delve into the divide-and-conquer relationship intrinsic to the GZM model (which naturally encompasses the RZM as its subset). Specifically, we question if operations within the GZM with large input arrays can be segmented into more digestible sub-tasks of a confined size. This consideration gains prominence when aiming to utilize hardware with a predetermined length to accommodate input arrays of undefined dimensions. As elucidated in Section-2.2, elementary tiling yields incorrect outputs. However, we can ascertain the following result: for any given GZM, any given length V (for instance, the SIMD vector length), any preset state of its state machine s, and any two input arrays a and b:

$$\begin{aligned} &\exists s', \Delta A, \Delta B, \Delta C \quad \text{such that :} \\ &GZM(s, a, b) = GZM(s, a[0:V], b[0:V])[0:\Delta C], \\ &+ + GZM(s', a[\Delta A : end], b[\Delta B : end]) \end{aligned} \tag{2.1}$$

Here, ++ symbolizes array concatenation. It's worth noting that the first component on the equation's right side possesses a bounded input, while the second component can be

continually divided following this method until all the sub-tasks are restrained by the size V.

To comprehend why this relationship is valid, let's visualize the execution of $GZM(s, a, b)$ using an illustrative approach: Imagine using paper cards marked with a "?" symbol to obscure and conceal the numbers of arrays a and b starting from their $V+1$-th elements. This scenario is depicted in Figure-2.11-left. As we proceed with the GZM execution, the moment it encounters the first "?", it halts as the next steps become undefined. At this juncture, we establish a breakpoint. Subsequently, we note down:

- The number of elements consumed in both input arrays, represented as ΔA and ΔB.
- The count of elements directed to the output, termed as ΔC.
- The prevailing state of the state machine, denoted as s'.

This is illustrated in Figure-2.11-right.

Following this, we discard all the paper cards and resume the execution until its conclusion. It's evident that the introduction and removal of these cards won't alter the GZM's output, but it essentially divides the execution into two distinct phases. Regarding the first phase, the outcomes of the following are identical:

1. The first ΔC elements resulting from $GZM(s, a[0 : V], b[0 : V])$.
2. The first ΔC elements emanating from $GZM(s, a, b)$.

For the second phase, the outcomes of these are analogous:

1. The result of $GZM(s', a[\Delta A : end], b[\Delta B : end])$.
2. The subarray of $GZM(s, a, b)$ commencing from the $(\Delta C + 1)$-th element to its termination.

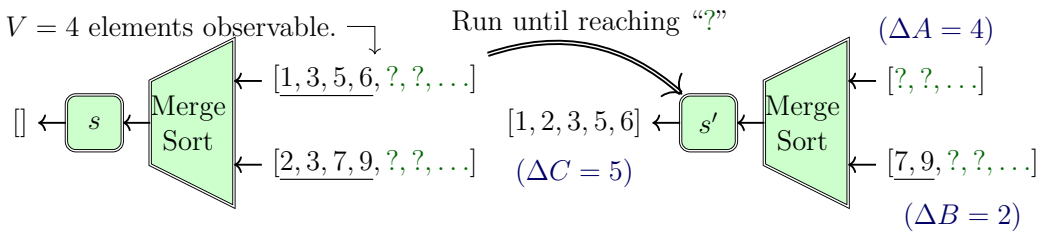
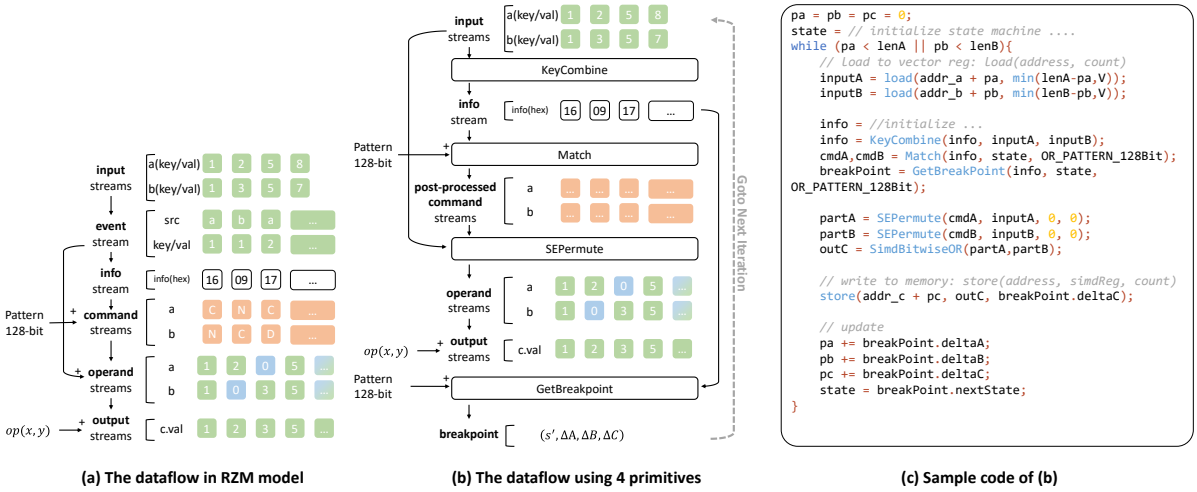


Figure 2.11: Visualization of the hypothetical execution process.




```
pa = pb = pc = 0;
state = // initialize state machine ....
while (pa < lenA || pb < lenB){
    // Load to vector reg: load(address, count)
    inputA = load(addr_a + pa, min(lenA-pa,V));
    inputB = load(addr_b + pb, min(lenB-pb,V));

    info = //initialize ...
    info = KeyCombine(info, inputA, inputB);
    cmdA,cmdB = Match(info, state, OR_PATTERN_128Bit);
    breakPoint = GetBreakPoint(info, state,
OR_PATTERN_128Bit);

    partA = SEPermute(cmdA, inputA, 0, 0);
    partB = SEPermute(cmdB, inputB, 0, 0);
    outC = SimdBitwiseOR(partA,partB);

    // write to memory: store(address, simdReg, count)
    store(addr_c + pc, outC, breakPoint.deltaC);

    // update
    pa += breakPoint.deltaA;
    pb += breakPoint.deltaB;
    pc += breakPoint.deltaC;
    state = breakPoint.nextState;
}
```

Figure 2.12: The correspondence between the RZM model and its implementation using the four primitives

Merging these two phases furnishes us with Equation-2.1.

2.4 Four Primitives to Implement RZM

We introduce four vectorized primitives for implementing RZM: `KeyCombine`, `Match`, `SEPermute`, and `GetBreakpoint`. Figure 2.12 contrasts the dataflow of RZM in part (a) with the implementation based on these four primitives in part (b). These primitives allow for the completion of an RZM execution in a single pass, consuming both input streams a and b to simultaneously generate the output stream $c.val$. Intermediate streams defined in the RZM model, such as the event stream, the two command streams, and the

two operand streams, are also immediately consumed once they are produced. All of these streams are consumed/produced at vector granularity, i.e., approximately V elements per iteration. The mapping of the RZM model to these primitives can be described as follows (at a high-level approximation):

1. The generation of the event stream from the two input streams is accomplished by `KeyCombine`. Essentially, it performs a merge sort.

2. The two command streams are generated from the event stream based on a given matching pattern (encoded in 128-bit). This is achieved using `Match`, which fundamentally executes multiple "32-entry 4-bit/entry" table lookups independently and in parallel.

3. The generation of the two operand streams from both the event stream and the command streams is accomplished by `SEPermute`. This process is conceptually akin to a `permute/shuffle` instruction.

4. The output stream ($c.val$) is generated from the operand streams using the binary operator $op(x, y)$. This can be achieved using traditional element-wise SIMD arithmetic primitives (suitable for CPU/GPU) and does not necessitate new primitives.

5. The divide-and-conquer parameter as mentioned in Section 2.3-3 (s', ΔA, ΔB, and ΔC) is generated by `GetBreakpoint`.

The aforementioned design is a reasonable approach, but it possesses two limitations. Firstly, we can improve efficiency by avoiding the explicit construction of the entire event stream. To understand this, consider that the command generation in the `Match` operation uses only 5 bits for each sliding window of the event stream. Moreover, `SEPermute` can directly generate the output stream from the input streams, eliminating the need

for the event stream. Thus, constructing the full event stream becomes superfluous. Secondly, within `SEPermute`, the command streams necessitate additional preprocessing before being processed by its permutation circuit. Given that `SEPermute` is often invoked multiple times with the same command, it is more efficient to transfer this recurring preprocessing task from `SEPermute` to its predecessor operation, `Match`. This is because `Match` is called only once per iteration, making it more suitable for post-processing tasks. Consequently, our pragmatic design introduces additional optimizations to the initially described approach as follows:

1. `KeyCombine` will no longer yield a complete event stream as its output; it will produce the info stream instead.

2. `SEPermute` will directly access the input streams, bypassing the event stream.

3. `Match` will utilize the info stream as input, replacing the event stream. Additionally, `Match` has an added responsibility: to post-process the command stream so that it can be seamlessly integrated into the permutation circuit of `SEPermute`.

2.4.1 KeyCombine

The `KeyCombine` function takes two input streams and internally performs a merge sort to produce the event stream. It then extracts the "5-bit" information from each sliding window, which includes determining the source of events (either 'a' or 'b') and comparing whether the keys of neighboring pairs are equal.

Use of the Bitonic Sorting Network: A well-established method for merge sorting two pre-sorted arrays (let's call them a and b) of fixed length V (which is the hardware vector length) in parallel is the bitonic sorting network. One issue with the traditional design of the bitonic sorting network is that it doesn't guarantee stability in the results: that is,

when elements have identical keys, they should be sorted first by their source (elements from a first, followed by those from b) and then by their original order in the input array. This stable property is a requirement for our GZM/RZM model. A straightforward solution is to include the source and the element's position in the original stream as part of a proxy key for comparison. For instance, the second occurrence of the number 2 in the input array $a = [1, 2, 2, 3]$ would have a proxy key represented as:

$$\text{proxy key} = \langle \text{key} = 2, \text{src} = \text{`a"}, \text{loc} = 2 \rangle$$

Lexicographical Sorting with Tuple Keys: There are scenarios where the keys are tuples comprising multiple fields. Applying the classical bitonic sorting network field-by-field yields individual results instead of a lexicographically sorted array. We address this by cascading the comparison outcomes between the processing of distinct fields. From a programmer's standpoint, performing a merge sort on tuples with k fields can be accomplished using k `KeyCombine` calls:

```
info = // initialize
info = KeyCombine(info, a.key.field1, b.key.field1)
info = KeyCombine(info, a.key.field2, b.key.field2)
...
info = KeyCombine(info, a.key.fieldk, b.key.fieldk)
```

Here, `field1` is the most significant field, while `fieldk` is the least significant.

The foundational architecture still relies on the bitonic sorting network. However, we have enhanced each switch with a 2-bit state. Remember that a classical bitonic sorting network is a $\log(2V)$-layer butterfly-like network, where each crossover functions as a 2-input 2-output compare-and-swap switch. There are a total of $V \log(2V)$ such switches, with each acting like a minimal sorter for 2 elements. The additional 2-bit state lets each

switch recall previous results. If all earlier (more significant) fields of the key are equal, the switch remains in an undecided state, with the determination deferred to the latter (less significant) fields. Conversely, if in a frozen state, subsequent fields can't modify the comparison outcome. We delineate four states using these 2 bits: Strong ¡, Strong ¿ (the two frozen states), and Weak ¡, Weak ¿ (the two undecided states). For each `KeyCombine` action, every switch (with proxy key inputs x and y) updates its state as follows:

1. In a Strong ¡ or Strong ¿ state, it remains unchanged.

2. In a Weak ¡ or Weak ¿ state, if $x.key \neq y.key$, the state transitions to Strong ¡ or Strong ¿, contingent on whether $x.key < y.key$ or vice versa.

3. In a Weak ¡ or Weak ¿ state, if $x.key = y.key$, it shifts to Weak ¡ or Weak ¿, dependent on whether $\langle x.src, x.loc \rangle < \langle y.src, y.loc \rangle$ or the reverse.

The rest mirrors the traditional bitonic network. The state of the switch is also incorporated into the `info`. Summing up, `info` now comprises: 1) the 2-bit states for each of the $V \log(2V)$ switches, 2) the "5-bits" gathered from every sliding window.

Lastly, following the lexicographical merge sort, `KeyCombine` needs to extract the "5-bits" from each sliding window. The "src" can be derived from the sorted proxy keys of the final `KeyCombine` execution. However, when juxtaposing a key with its adjacent key, it must aggregate the "equals" through the logical AND across all previous k `KeyCombine` executions. This is because two keys aren't equal if even one of their k fields differs.

2.4.2 SEPermute

`SEPermute` is a vital component designed to generate the operand streams from the input stream. Functionally, it mirrors the semantics of traditional data permute or

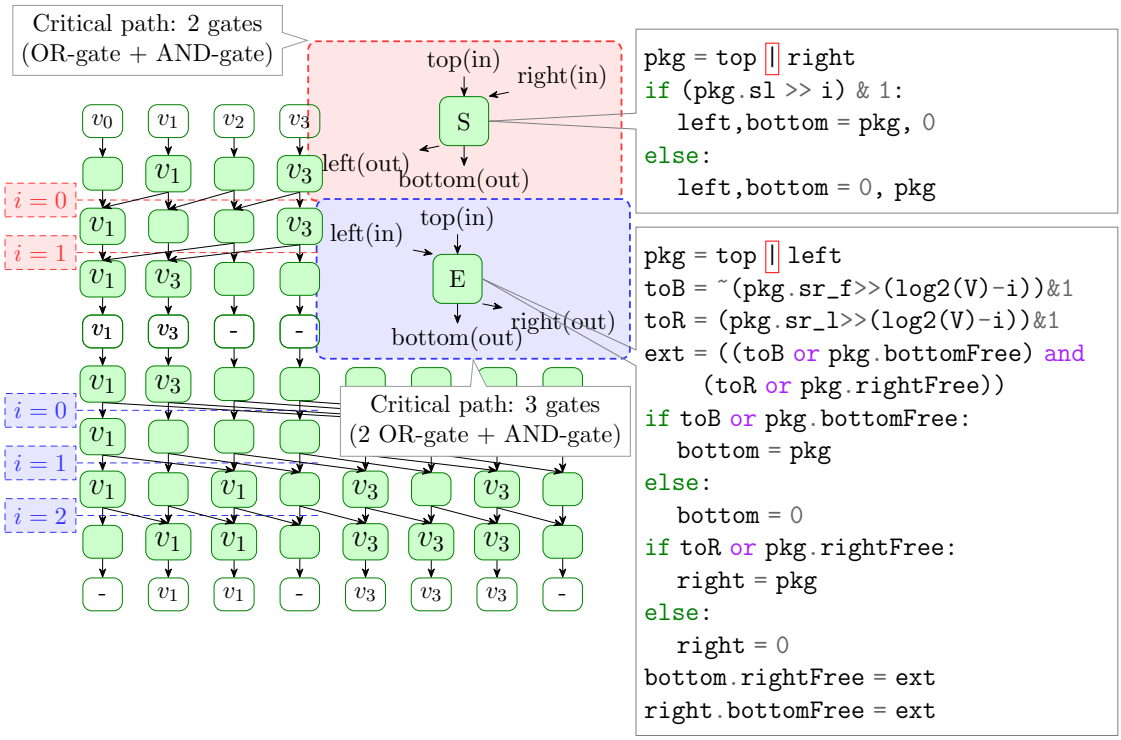


Figure 2.13: Illustration of the Squeeze-Expand network.

shuffle instructions. Here, the post-processed command streams emulate the control masks typically found in conventional permute/shuffle instructions.

Order-Preserving Permutations Only: The permutation functionality demanded by our problem is strictly of an order-preserving nature. That is, elements in the output retain the order they had in the input. Valid operations include element removal, insertion of constant elements (e.g., 0), and element duplication. Changing the original order, however, is not permitted. Taking an input such as [1,2,3,4], valid outputs might resemble [1,0,0,2,2,2,0,4], but not [4,2,3,1]. This specificity suggests that the conventional $O(V^2)$-cost crossbar approach is excessive for our needs.

The Squeeze-Expand Network Solution: To cater to the unique order-preserving permutations required, we introduce the Squeeze-Expand network, an efficient $O(V \log(V))$-cost solution. The network is structured with $\log(V)$ squeeze layers, followed by $\log(2V)$ expand layers. A visual representation of this design is presented in Figure-2.13.

The squeeze network can drop part of the elements and shift the remaining elements towards the left to make them compact. For example, [1,2,3,4] → [1,3,-,-]. At layer i (counting from 0), the elements can either shift left by 2^i positions or move straight down. Such usage of the squeeze network is not a new idea and was commonly used for stream compaction.

The expand network can insert an extra constant element between existing elements or duplicate elements. For example, [1,3] → [0,1,1,0,3,3,3,0]. It appears like an upside-down version of the squeeze network. At layer i, the elements can 1) shift right by $V/2^i$ positions, 2) move straight down, or 3) broadcast in both directions. The last case is used to duplicate elements. As far as we know, this is the first time such a network with duplication support has been used.

The combination of the squeeze network and the expand network is sufficient to cover all order-preserving permutations. The primary challenge is to implement the routing control logic. We first define the format of the control signals (the post-processed command stream) it receives from `Match`:

1. For each element, there is a valid bit that decides whether it will appear in the output.

2. For each element, there is a number specifying how many positions it will shift to the left in the squeeze network in total. It is a $\log(V)$ bit integer, named `sl` ("shift left").

3. For each element, it will shift to the right and be broadcasted over a range in the expand network. There are two numbers specifying the amount of right-shift for both the first and last elements of the broadcasted range. If no duplication is made, then the numbers for the first and the last will be equal. These two numbers are $\log(2V)$ bit integers, named `sr_f` and `sr_l` ("shift right first/last").

The complete control signal for each element is therefore a tuple comprising 4 components: (`valid`, `sl`, `sr_f`, `sr_l`).

Routing at squeeze network: At layer i, the squeeze network uses the i-th bit ($i = 0$ being the least significant bit) of `sl` to decide whether it should shift to the left. The correctness is based on a simple fact: $\text{sl} = \text{sl}[0] \times 2^0 + \text{sl}[1] \times 2^1 + \text{sl}[2] \times 2^2 + \ldots$, where $\text{sl}[b]$ is defined as `(sl>>b) & 1`.

Routing at expand network: At layer i, the expand network forwards elements to the right by $V/2^i$ positions, straight down, or both (for duplication). Assume one copy of the element has been shifted to the right by x positions (thus, $\texttt{sr\_f} \leq x \leq \texttt{sr\_l}$). The routing rules are:

1. Forward to the right if $x + V/2^i \leq \texttt{sr\_l}$.
2. Forward straight down if $\texttt{sr\_f} \leq x$.

There is an efficient way to implement the above rules without using adders and comparisons, which is advantageous in hardware to reduce cost and latency. As shown in Figure-2.13, we use two extra flag bits (`bottomFree` and `rightFree`) that are initialized to 0 to incrementally build the above inequalities layer-by-layer, thus reducing the latency of each layer to 3 gates.

2.4.3 Match

`Match` first generates two command streams (for both a and b) from the info stream and then postprocesses them. The initial step involves using each "5-bit" to look up a 32-entry table with 4-bits per entry, encoding the matching pattern. This step is embarrassingly parallel. The subsequent step transforms the `command` streams into the (`valid`, `sl`, `sr_f`, `sr_l`) tuples, as mentioned in `SEPermute`. This transformation is essentially a set of prefix

sum operations. We have two streams, a and b. Since they are symmetric, we can denote them as $(x = a, \bar{x} = b)$ or $(x = b, \bar{x} = a)$. The postprocessed command streams are referred to as postCmd streams.

1. (About `valid`) Whether an element is used in the final result depends solely on whether it is pushed; i.e., the command is PushThis in the command stream x and src is x.

2. (About `sl_l`) The number of positions an element should shift to the left is the sum of preceding elements that were not pushed to the array. That is, the count of prior elements whose command is NOT PushThis in the command stream x and src is x.

3. (About `sr_f`) The number of positions to shift an element in x to the right is the count of extra elements inserted before this element either by its opposite side $\bar{x}$—i.e., the count of prior elements whose command is NOT NoPush in command stream x and src is $\bar{x}$—or by the same side when inserting default elements, i.e., the count of prior elements whose command is PushDefault in command stream x and src is x.

4. (About `sr_l`) The number of duplications (`sr_f-sr_l`) corresponds to the number of PushMostRecent commands executed for this element. Specifically, it's the count of directly subsequent elements whose command is PushMostRecent in the command stream x, not interrupted by any other commands like PushThis or PushDefault.

We observe that "counting the number of prior elements that satisfy a certain condition" across all positions is a parallel prefix sum operation, which addresses points (2) and (3). Similarly, point (4) can be understood as a parallel segmented suffix sum. Thus,

determining the tuple (`valid`, `sl`, `sr_f`, `sr_l`) for each element can be efficiently implemented in hardware using established techniques. One distinction we must account for is the presence of V results for a and another V for b. These $2V$ results are intertwined in a merge-sorted order. To separate them, we can route them along the inverse path in the bitonic network (the necessary routing data is already available: the 2-bit states of comparators are encapsulated within `info`). An additional circuit-level enhancement is to integrate the prefix sum operation into the separation step, allowing both processes to be completed in a single pass.

2.4.4 GetBreakpoint

The `GetBreakpoint` calculates the divide-and-conquer parameters as outlined in Section-2.3.3, specifically $(s', \Delta A, \Delta B, \Delta C)$. As per its definition, the initial step involves determining the moment when our GZM/RZM can no longer progress due to the uncertainty of the value of the element following the first V elements of both input arrays. It's crucial to understand that the last event executed before this breakpoint corresponds to one of the two V-th elements of the input arrays, specifically the one that appears earlier in the event stream. This determination can be made solely using the event stream. However, in our design, the `GetBreakpoint` accepts the more concise info stream as its input. To ascertain the values of $\Delta A, \Delta B$, and ΔC, the `GetBreakpoint` module needs only to tally the number of consumed and pushed elements as indicated in the command stream up to the last event executed before the breakpoint. Lastly, representing the state s' accurately demanded a slew of engineering tactics and adaptations. While we won't delve into the specifics here, the pivotal outcome is our ability to represent s' with just two bits.

2.5 Evaluation

In this section, we evaluate the correctness, performance, and flexibility of our design. We have implemented several representative workloads, which can be categorized into two groups: the simple workloads that are direct MSOs, and the complex workloads where an MSO serves as a foundational building block for more intricate operations. We compare the baseline CPU with the same CPU enhanced by our SIMD extension. Essentially, we contrast two optimized kernels: one employs only traditional scalar and SIMD instructions, while the other also leverages the proposed SIMD primitives. For all workloads, we employ SIMD baselines where appropriate and resort to scalar baselines otherwise. As mentioned in Section-2.2, only two special cases of MSO can be expressed in SIMD form using existing SIMD primitives. We defer the examination of the other two use-cases of our design, namely "baseline GPU vs. the same GPU with our SIMD extension" and "MSO-related accelerators vs. the same accelerator with our $O(V \log(V))$-cost design as a drop-in replacement", to future research.

2.5.1 Experiment Setup

We extend the GCC(v10.1.0)/Binutils(v2.35) toolchain and the gem5(v20.1.0.2) CPU simulator to support our proposed SIMD primitives. The base ISA employs Armv8 with the SVE extension. The simulated processor is an out-of-order CPU modeled after the Arm Cortex A15. Additionally, a 32KB L1I cache, 32KB L1D cache, 1MB L2 cache, and a TaggedPrefetcher for the L1D cache are enabled. Given that instruction latency in cycles is influenced by numerous factors such as the technology node, frequency, and voltage, we have conservatively estimated the latency of the proposed primitives. This estimation is based on a straightforward reference: "a 32-bit integer addition can be executed in one cycle". Based on this, we established the cycle count for instruction latency. To derive

an even more reliable lower bound for performance improvement, we further scaled the latency for each primitive by a factor of approximately 3, ensuring ample margin for real-world implementation[2]. Lastly, given a SIMD width V, the latency for our proposed primitives in the gem5 simulator is detailed in Table 2.4. The area synthesized at 22nm is depicted in Table-2.5. For context, a 4-core Intel Haswell processor fabricated at a 22nm technology node has an area of around 200mm2. Incorporating our primitives (with $V = 16$, for instance) results in a mere 0.098% area overhead.

Discussion: Practical Engineering Details. Firstly, the encoded primitives necessitate bit-packing of their input/output to ensure the number of input register operands does not exceed three and that the number of output registers remains at one. Secondly, even though the `KeyCombine` primitive exhibits a relatively long latency, a sequence of cascaded `KeyCombine` primitives designed to manage multi-field keys can be fully pipelined. This is possible even when data dependencies exist on `info`. The reason being, different segments of `info` are updated exclusively in one of the $\log(2V)$ layers of the bitonic network and these updates occur in non-overlapping cycles. Consequently, cascaded `KeyCombine` primitives can be pipelined layer-wise.

Table 2.4: Latency of primitives.

Primitive	Latency(cycles)
KeyCombine	$3\log_2(2V)$
Match	$\log_2(2V)$
SEPermute	$\log_2(2V)$
GetBreakpoint	$\log_2(2V)$

Table 2.5: Area overhead 22nm

V	4	16	64
Area(mm^2)	7.83E-03	4.91E-02	2.74E-01

[2]Excluding this scaling factor, there would be an approximate 20% additional performance boost.

2.5.2 Simple Workloads

Figure 2.14 displays the throughput for set operations, including set-union, set-intersection, set-diff, and set-xor, as well as database join operations, namely join-inner, join-outer, join-diff, join-xor, and join-left. These kernels showcase the flexibility of our method in supporting various matching patterns. Meanwhile, Figure 2.15 exhibits the throughput of element-wise additions and multiplications for both real and complex numbers between sparse vectors, matrices, and tensors, as well as an MSO associated with merging adjacency lists in graph analytics applications. These kernels underline our method's adaptability to handle different tuple sizes for keys and values, and varying functions of $op(x, y)$. It is assumed that all inputs are already sorted into a key-value array format. In our simulations, both len(a) and len(b) are set to 10k, with throughput defined as $(len(a) + len(b))/time$. We have also marked the speedup compared to the scalar baseline[3] for three SIMD widths, $V = 4, 16, 64$, which correspond to SIMD bitwidths of 128-bit, 512-bit, and 2048-bit, respectively. The average speedup across all these kernels for $V = 16$ is $7.1\times$.

2.5.3 Breakdown of Speedup

We analyze the sources of speedup by representing the execution time of an MSO in terms of three factors:

[3]Of all the aforementioned kernels, currently only set-intersection has a known SIMD implementation method, so only the scalar baseline is used here.

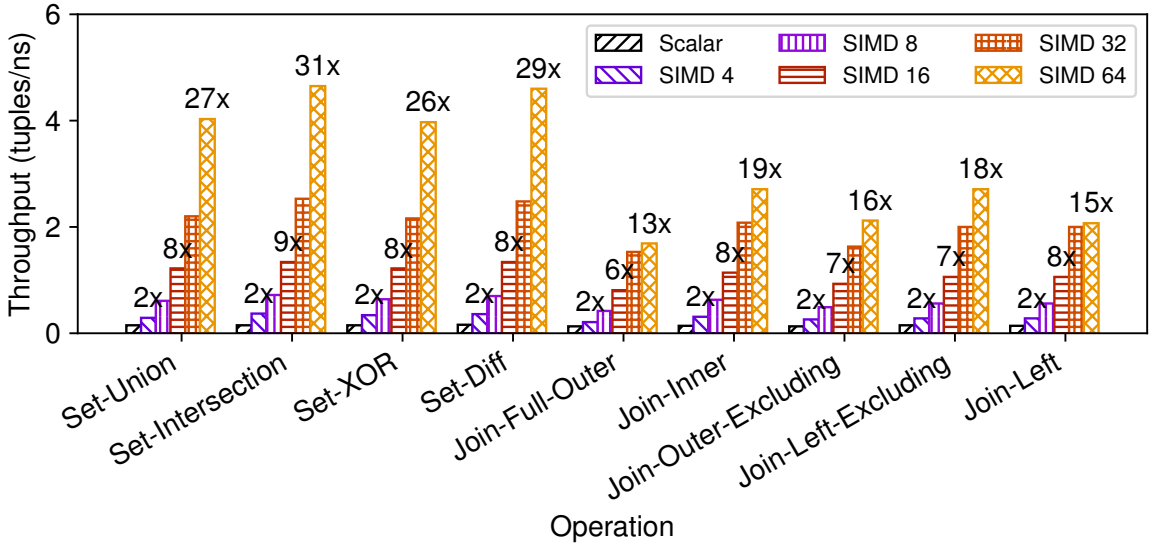


Figure 2.14: Throughput of set and join operations.

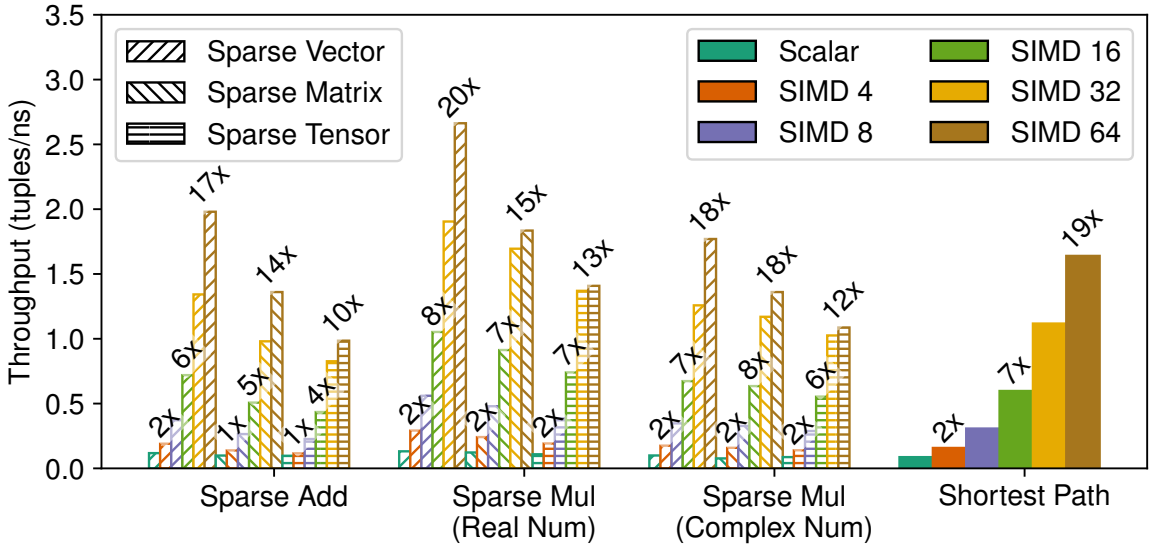


Figure 2.15: Throughput of sparse vector/matrix/tensor operations and graph operations.

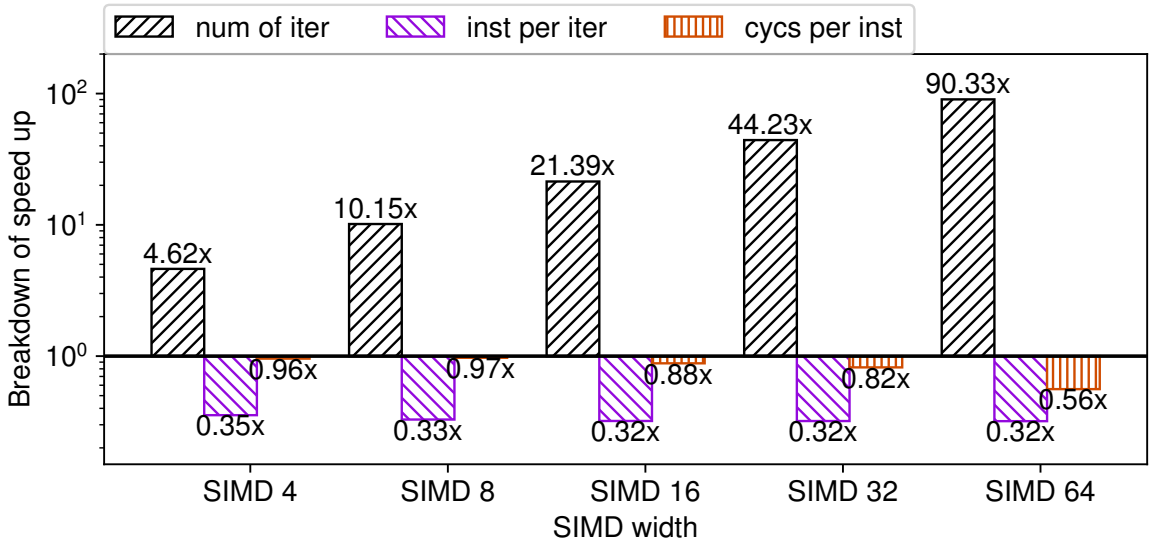


Figure 2.16: Breakdown of speedup: SIMD vs. scalar.

$$\begin{aligned}\text{Time (in cycles)} = \text{number of iterations} \\ \times \text{ instructions per iteration} \\ \times \text{ cycles per instruction.}\end{aligned}$$

Figure 2.16 illustrates how SIMD impacts the three aforementioned factors, using sparse vector addition as an example. From the figure, several observations can be made. When compared to the scalar implementation, the SIMD implementation:

1. Reduces the number of iterations by approximately $1.5V$ times ☺, which is the primary driver behind the throughput improvement.

2. Incurs about 3 times more instructions per loop ☹, attributed to the increased complexity.

3. Experiences a higher cycles-per-instruction (CPI) ☹due to a combination of reasons.

Specifically, while the SIMD implementation gets rid of the difficult-to-predict branch present in scalar codes ☺, it also faces longer instruction latency and is more susceptible to bandwidth pressure ☹. The final speedup is a product of these three factors.

Generally, the throughput scales favorably with the SIMD width V up to $V = 32$. The growth in throughput stagnates when V is extended to 64, a phenomenon attributable to the memory subsystem becoming saturated. At present, the longest SIMD width in mainstream CPUs is 16 (as seen in AVX-512). Therefore, the saturation issue observed at $V = 64$ remains hypothetical. Furthermore, in real-world scenarios, CPU manufacturers typically enhance the memory subsystem bandwidth (especially cache bandwidth) whenever they opt to double their SIMD width, which would likely mitigate the bandwidth constraints.

2.5.4 Complex Workloads

We assess two intricate workloads that are built upon MSOs: 1) sorting, and 2) general sparse matrix-matrix multiplication (SpGEMM).

Sorting. For our baselines, we employ two scalar sorting algorithms: quicksort (via `std::sort`) and merge sort (via `std::stable_sort`), as well as a SIMD sorting algorithm, specifically Bramas SIMD[16]. The benchmarks encompass both 32-bit integer sorting and 32-bit key-value sorting. We use arrays that are randomly generated, each containing 10k elements. Figure 2.17 illustrates the absolute performance of the baseline methods in comparison with our design (i.e., the proposed SIMD). When $V = 16$, our approach is 3.4× swifter than Bramas SIMD and 8.0× more rapid than the scalar sorting methods.

The efficiency of our proposed SIMD approach is not just demonstrated by its speed

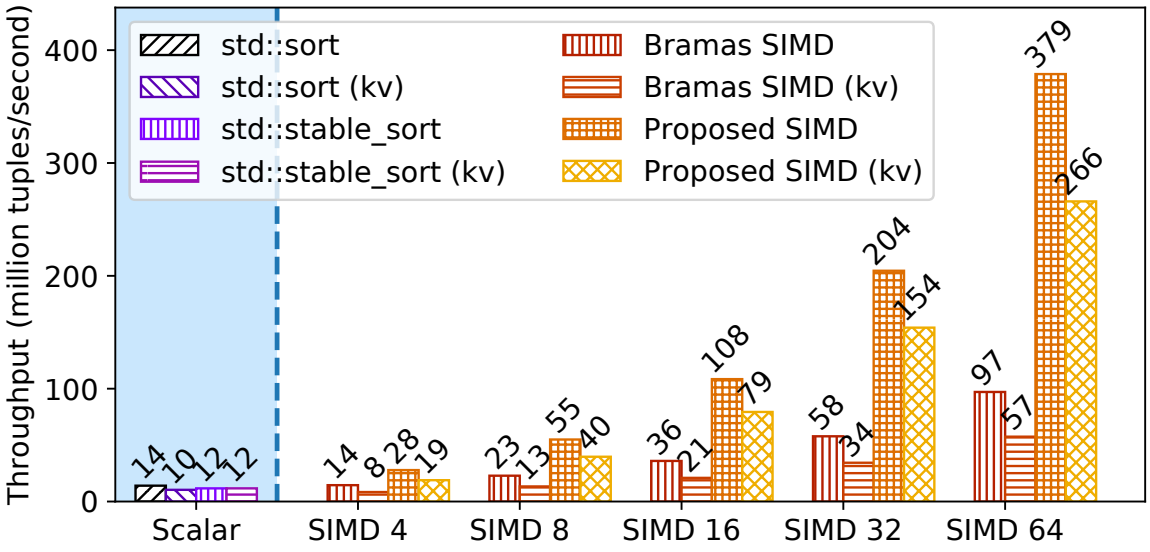


Figure 2.17: Throughput of sorting algorithms.

Table 2.6: Properties of 4 sorting methods.

	`std::sort`	`std::stable_sort`	Bramas	Proposed
SIMD	✗	✗	✓	✓
Stable	✗	✓	✗	✓

but also by its stability, as highlighted in Table 2.6. A sorting algorithm is considered stable if the relative order of key-value pairs with identical keys remains unchanged post-sorting. Notably, traditional algorithms like Quicksort and other SIMD sorting methods, including bramas, are typically unstable. A prevalent remedy to ensure stability in these methods is to embed the original position as a secondary key. However, this workaround induces additional overhead, leading to an approximately 1.5× surge in execution time[17].

The introduction of the `GetBreakpoint` primitives offers two distinct benefits over the conventional loop-tiling technique: (1) it halves the iteration count, and (2) it ensures stability. As depicted in Table 2.7, the rudimentary tiling approach for sorting processes $2V$ elements each iteration but only yields V elements. In contrast, the `GetBreakpoint` mechanism produces roughly $1.78V$ elements on average per iteration when $V = 16$.

Table 2.7: Progress per iteration using two tiling methods.

	SIMD 4	SIMD 8	SIMD 16	SIMD 32	SIMD 64
Naive	4 (V)	8 (V)	16 (V)	32 (V)	64 (V)
GetBreakPoint	6.16 ($1.54V$)	13.53 ($1.69V$)	28.53 ($1.78V$)	59.00 ($1.84V$)	120.48 ($1.88V$)

SpGEMM. General sparse matrix-matrix multiplication (SpGEMM) can often be realized as the weighted sum of sparse vectors. Such an operation can particularly benefit from the accelerated sparse vector addition, facilitated through an OR-pattern MSO using SIMD. To evaluate the end-to-end performance improvement, we implemented a SpGEMM kernel drawing upon the SIMD-optimized sparse vector addition. This was pitted against an assortment of scalar baselines for comparison.

As Figure 2.18 illustrates, both the throughput and the relative speedup (for SIMD widths of 4, 16, and 64) versus the finest scalar baseline for each dataset are presented. Remarkably, for a SIMD width of $V = 16$, an average speedup of 4.4× is attained.

Shifting focus to Figure 2.19, it charts out the fraction of the total SpGEMM execution time consumed by the MSO. Here, the innermost narrow rings correspond to diverse datasets, while the outermost expansive rings depict their geometric mean. One key takeaway is the overwhelming dominance of the MSO in the execution timeline, even when leveraging broad SIMD units — for instance, it constitutes 90% at $V = 16$ (512-bit). Moreover, the speedup achieved through SIMD over standalone sparse vector addition (as spotlighted in Figure 2.15) is predominantly sustained in holistic SpGEMM kernels. For SIMD widths of $V = 32$ and $V = 64$, Amdahl's law mildly impacts the performance, as the non-MSO segments (or the residual parts) account for $14.5\% \sim 19\%$ of the execution duration.

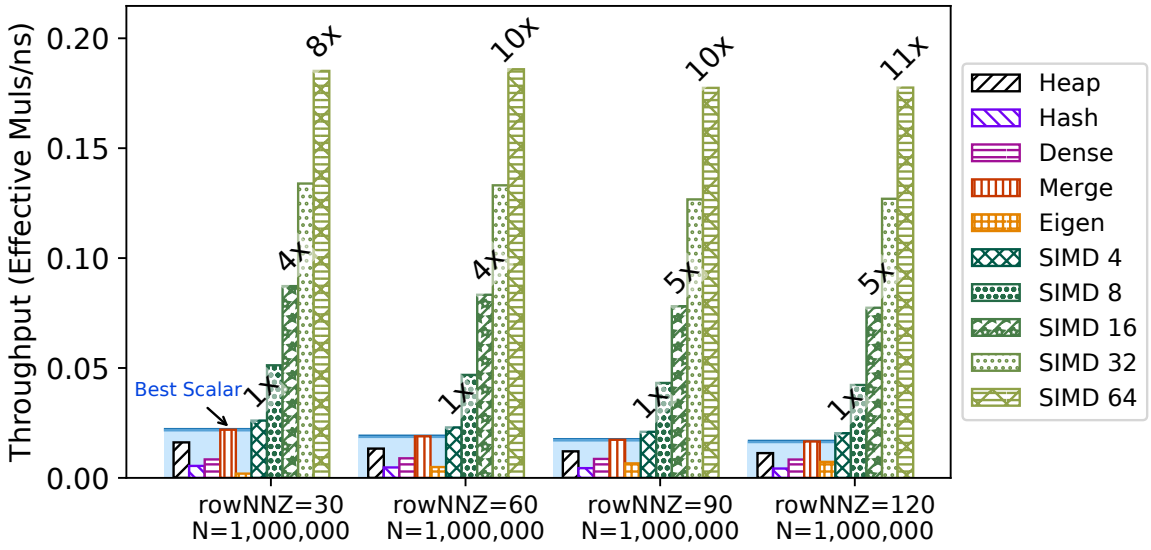


Figure 2.18: Throughput of SpGEMM and relative speedup of SIMD 4/156/64 vs. the best one of scalar.

Quality Assurance for SpGEMM Baselines. In our pursuit of a reliable benchmark for SpGEMM, we first gravitated towards the renowned and portable library, Eigen[18]. A pertinent issue is the platform specificity of many well-optimized open-source libraries and

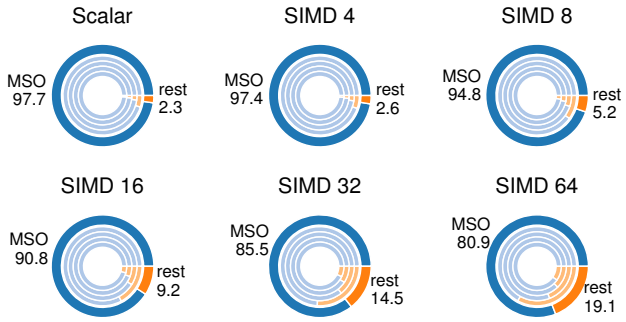


Figure 2.19: Percentage of execution time spent on the MSO in SpGEMM under different SIMD width.

research artifacts—most are tailored for x86, while our focus is on Armv8. Consequently, a direct runtime comparison on our simulated Arm core is unfeasible.

To navigate this challenge, we meticulously implemented all major SpGEMM algorithms documented in the literature in a platform-agnostic manner. This includes the heap, hash, dense vector, and iterative merging algorithms. Following optimization, these algorithms were executed on both x86 and Arm platforms, establishing a conduit for inter-platform comparison. As depicted in Figure 2.20, our baseline set closely mirrors the performance of contemporary, state-of-the-art SpGEMM implementations such as Yu's Hash and HashVec[19]. Notably, these implementations already outperform MKL and prior work. This congruence in performance bolsters our confidence that our proposed design would exhibit comparable speedups over x86-specific libraries, albeit through an indirect comparison. It's worth highlighting that while Yu's HashVec does employ SIMD, its application is confined to parallel hash table probing. This restricts its potential benefits from broader SIMD widths in future implementations.

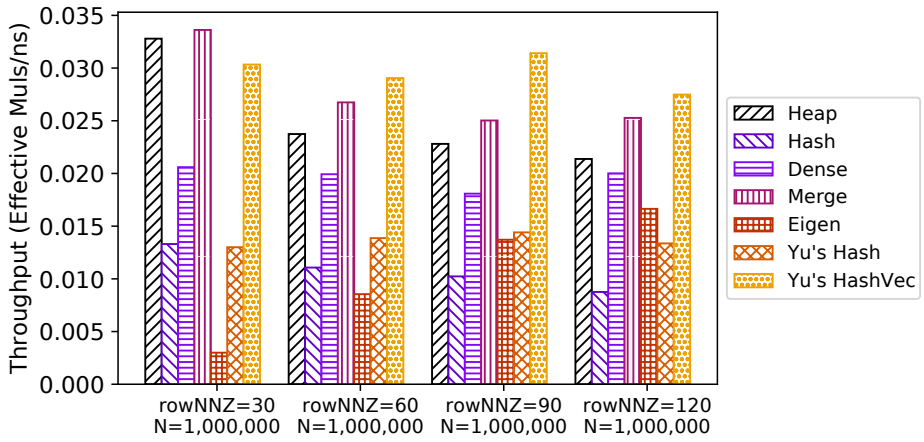


Figure 2.20: Performance comparison of our platform-agnostic baseline collection on x86 with x86-specific SpGEMM libraries (Yu's Hash and HashVec).

2.6 Conclusion

In this chapter, we present a one-for-all solution to implement MSOs in parallel and overcome the limitations of prior work. We note the following: (1) different MSO patterns can be encoded as functions that map a sliding window of three elements, post-merge sort, to actions; (2) the V-to-V comparison problem evident in prior work can be transformed into a sorting problem; (3) a comparator of fixed size can compare tuples of any size lexicographically, provided it is equipped with a 2-bit state machine. Our techniques enable a single hardware unit to support up to 2^{128} patterns with complete flexibility in datatype and value calculation using only $O(V \log(V))$ hardware resources. In our evaluations on CPUs using the gem5 simulator, for $V = 16$ (512-bit SIMD, 32-bit element), we achieve significant speedups on a variety of representative kernels. This includes set operations (8.4×), database joins (7.3×), sparse vector/matrix/tensor additions and multiplications with real/complex numbers (6.5×), merge sort (8.0× compared to scalar, 3.4× compared to the state-of-the-art SIMD), and SpGEMM (4.4× over the best benchmark in our baseline collection).

Chapter 3

High-level Operations on Ordered (Key,Value) Data Structures

In real-world applications, many frequently used data structures/algorithms on ordered (key, value) pairs cannot be efficiently accelerated using existing SIMD instruction and have a more complex data organization than a simple sorted array. However, in this chapter, we will show how to rewrite those data structures/algorithms into a form dominated by merge style operation (MSO) and therefore obtain the performance advantage of SIMD using the same SIMD primitives introduced in the last chapter. This chapter complements the last chapter, showing the broad application of the proposed SIMD primitives. Following, we will discuss three operations: heap, binary search, and K-way additive merge.

3.1 Binary Heap based on MSO

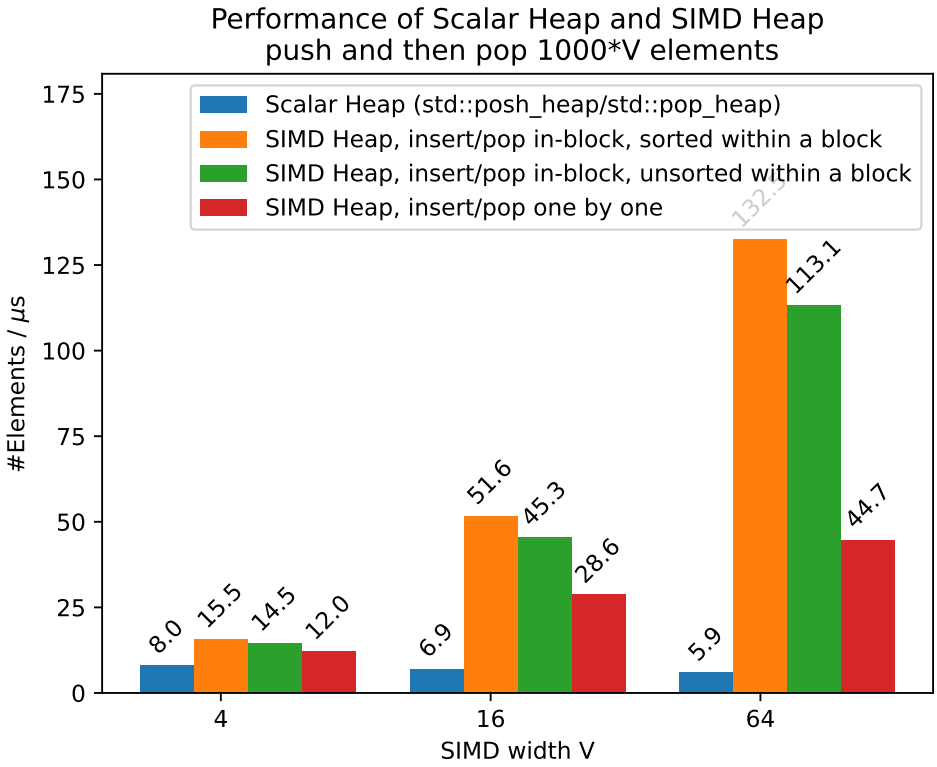


Figure 3.1: The performance of the SIMD heap is compared with that of the scalar implementation. The SIMD heap allows for any combination of insertion and extraction operations with different granularity and ordering properties, including: 1) a block of sorted data, 2) a block of unsorted data, 3) sorted data, and 4) unsorted data inserted one-by-one.

Traditional binary heaps are typically considered purely scalar, and it may not be immediately clear where SIMD instructions could be effectively applied, as a SIMD instruction operates on V elements simultaneously, where V is the SIMD width. However, we can rewrite the traditional binary heap as a pair of larger and smaller heaps, with the overall algorithm working by combining their results. The smaller heaps have a constant size and can be operated on in unit time. The larger heap contains all the remaining data, and operates at most $1/V$ as frequently as the smaller heap. However, each operation can be accelerated by a factor of V when using MSO SIMD primitives.

3.1.1 The Smaller Heap

The smaller heap is implemented as a short sorted array that contains fewer than V elements, where V is the SIMD width.

1. The heap insertion operation involves inserting the new element into the sorted array. The position for insertion can be easily determined using a SIMD scalar-vector comparison. If the array contains exactly V elements after the insertion, all V elements will be moved to the larger heap, and the smaller heap will be cleared.

2. The heap extraction operation involves extracting the smallest element from the sorted array, which is simply the first element. We can use SIMD vector copy to move the remaining elements to the front of the array.

3.1.2 The Bigger Heap

The bigger heap is implemented as a complete binary tree, where each node contains V elements. It enforces several constraints:

1. The V elements in each node are sorted.

2. The elements in child nodes are always equal to or greater than the elements in parent nodes.

3. The top node is a special case, where it may contain fewer than V elements. However, it must be non-empty unless the bigger heap is empty as a whole.

Insertions into the bigger heap are limited to batches of V elements that are already sorted. To insert a batch, a new node is created from the V elements and added to the end of the tree. The heap-fix process is then performed to restore the heap property, starting at the newly inserted node and proceeding up the parent chain. Let's denote the

current node as q, and its parent node as $p = \text{parent}(q)$. Using SIMD-accelerated merge, we exchange the $V + V$ elements in nodes p and q such that the elements in p are less than or equal to the elements in q. This means that after the merge, the parent node p will contain the smaller half of the $V + V$ elements, while the child node q will contain the larger half. The above process can be summarized in the following pseudocode:

$$
\begin{aligned}
p &\leftarrow \text{parent}(q) \\
[\text{low half}, \text{high half}] &\leftarrow \text{merge}(\text{Content}(p), \text{Content}(q)) \\
\text{Content}(p) &\leftarrow \text{low half} \\
\text{Content}(q) &\leftarrow \text{high half}
\end{aligned}
$$

The heap-fix process will continue until the root is reached, or as an optimization, terminate early if the smallest element in p is not changed during the exchange. Whether or not early termination is taken, the heap property is restored when the heap-fix steps end.

For heap extraction from the bigger heap, the first element in the top node is removed and returned. As with the smaller heap, we can use SIMD vector copy to move the remaining elements in the top node to the front. If multiple extractions result in the top node becoming empty, additional heap-fix steps are required to refill it. These steps consist of a top-down pass and a bottom-up pass.

The top-down pass treats the top node as a "hole" and moves it towards the bottom of the tree (the leaves) using a series of exchange steps. This process makes the binary tree incomplete, as one of the leaves will be missing. The bottom-up pass then restores the complete binary tree by moving the last leaf to the hole and performing heap-fix steps bottom-up to restore the heap property. These bottom-up steps are similar to those used in the insertion operation but are more likely to terminate early.

Here we will discuss the exchange steps to move the "hole" towards the bottom in more detail. Let's denote the current node (the "hole") as q, and its children nodes as c_1 and c_2. We compare the last element of the two children nodes, c_1 and c_2, and denote the smaller one as c_- and the larger one as c_+. For example,

$$\text{last}(c_1) \leq \text{last}(c_2) \implies c_- = c_1 \text{ and } c_+ = c_2 \tag{3.1}$$

$$\text{last}(c_1) > \text{last}(c_2) \implies c_- = c_2 \text{ and } c_+ = c_1 \tag{3.2}$$

We will then use SIMD-accelerated merge to process the $V+V$ elements. The current node (the "hole") q will take the smaller half of the $V+V$ elements, the child node c_+ will take the larger half, and the child node c_- will become the new "hole".

$$[\text{low half}, \text{high half}] \leftarrow \text{merge}(\text{Content}(c_-), \text{Content}(c_+)) \tag{3.3}$$

$$\text{Content}(q) \leftarrow \text{low half} \tag{3.4}$$

$$\text{Content}(c_+) \leftarrow \text{high half} \tag{3.5}$$

$$\text{Content}(c_-) \leftarrow \text{new hole} \tag{3.6}$$

The exchange steps will proceed with the new hole being c_- until the bottom is reached. The subsequent bottom-up pass has already been described in the previous discussions.

Mixing extraction and batch insertion can lead to a corner case where the top node is not full while the heap-fix step of batch insertion reaches the top. In this case, the merge will have fewer than $V+V$ input elements. This can be fixed by always letting the high half contain the larger V elements, and the low half contain the remaining elements, which may be fewer than V.

3.1.3 The overall algorithm

The overall algorithm for insertion and extraction involves forwarding the insertion and extraction to the smaller and bigger heaps, with proper coordination between the two sub-heaps.

To perform insertion, we first insert the new element into the smaller heap. The insertion procedure terminates if the smaller heap contains fewer than V elements. Otherwise, all the elements in the smaller heap will be moved to the bigger heap, and a batch insertion will be performed.

To perform extraction, we compare the top elements in the smaller heap and the bigger heap. We select the smaller one and perform an extraction operation on the corresponding heap.

Observation: The heap-fix operations at the bigger heap will occur at most $1/V$ as frequently as the scalar insertion/extraction operations in the overall algorithm.

This is because:

1. To trigger every heap-fix operation related to batch insertion, we need to insert at least V elements into the smaller heap to make it full.

2. To trigger every heap-fix operation related to extraction, we need to extract at least V elements from the bigger heap to make the top node empty.

This leads to two advantages compared to the textbook scalar binary heap:

1. The parallelism of SIMD is exploited: The heap-fix operation operates on nodes with a granularity of V elements per node, making the use of SIMD possible. Meanwhile, heap-fix will be triggered at $1/V$ of the original frequency, so the total computation is not increased. The overall performance is improved by exploiting the specialized SIMD extension to accelerate the $V + V$ element merge.

2. This design is more cache-friendly when the heap is large. Most insertion/extraction operations (actually $V-1$ out of V) will only touch the smaller heap and the top node of the bigger heap. When the heap-fix is performed in the remaining $1/V$ cases, all V continuous elements in a node are used, increasing the portion of useful data in a cache line. The total cache miss rate is eventually reduced by a factor of V.

Algorithm 1: Insertion to the heap

```
Data: Smaller heap: s is a sorted array of size n_s; ensuring always have n_s ≤ V.
Data: Bigger heap: B is a number of blocks of V elements. The n_B is the
      number of blocks. n_t is the number of elements in the top node.
Input: input element x
begin
    /* 1. insert x into the smaller heap using textbook insertion. Because
       n_s ≤ V, the following step finishes in one step using SIMD.        */
    last ← 0;
    // Use SIMD
    for i ← 1 to n_s do
        if s[i] ≤ x then
            last ← i + 1;
        else
            s[i + 1] ← s[i];
        end
    end
    s[last] ← x;
    n_s ← n_s + 1;
    /* 2. if the smaller heap is full, move all elements to the bigger heap
       and perform the batch insertion.                                    */
    if n_s = V then
        /* 2.1. move all elements to the bigger heap                      */
        // Use SIMD
        for i ← 1 to V do
            B[n_B][i] ← s[i];
        end
        n_B ← n_B + 1;
        FixHeap(B, n_B, n_B − 1, n_t);
        n_s ← 0;
    end
end
```

Algorithm 2: Fix Heap Bottom Up

```
Function FixHeap(B, n_B, at, n_t) is
    while at > 0 do
        parent ← (at − 1)/2;
        if parent = 0 then
            n_p ← n_t;
        else
            n_p ← V;
        end
        // Use SIMD
        merged ← Merge_Sort(B[parent], B[at]);
        // Use SIMD
        for i ← 1 to n_p do
            B[parent][i] ← merged[i];
        end
        // Use SIMD
        for i ← 1 to n_t do
            B[at][i] ← merged[n_p + i];
        end
        at ← parent;
    end
end
```

Algorithm 3: Extration from the heap

```
Data: Smaller heap: s is a sorted array of size n_s; ensuring always have n_s ≤ V.
Data: Bigger heap: B is a number of blocks of V elements. The n_B is the
      number of blocks. n_t is the number of elements in the top node.
Output: the top element y
begin
    /* 1.  compare the top elements in the smaller heap and the bigger heap   */
    topAt ← null;
    if n_s ≠ 0 then
        topAt ←' small';
    end
    if  topAt ←' big';
     then n_t > 0 and B[0][0] < n_s[0]
    end
    /* 2.  perform the extraction on the corresponding heap                   */
    if topAt = null then
        return null;
    end
    else if topAt =' small' then
        /* 2.1.  extract from the smaller heap                                */
        y ← s[0];
        n_s ← n_s − 1;
        // Use SIMD
        for i ← 1 to n_s do
            s[i − 1] ← s[i];
        end
        return y;
    end
    else if topAt =' big' then
        /* 2.2.  extract from the bigger heap                                 */
        y ← B[0][0];
        n_t ← n_t − 1;
        // Use SIMD
        for i ← 1 to n_t do
            B[0][i − 1] ← B[0][i];
        end
        if n_t = 0 then
            // Use SIMD
            FixHeapTopDown(B, n_B, 0);
        end
        return y;
    end
end
```

3.2 Batched Binary Search based on MSO

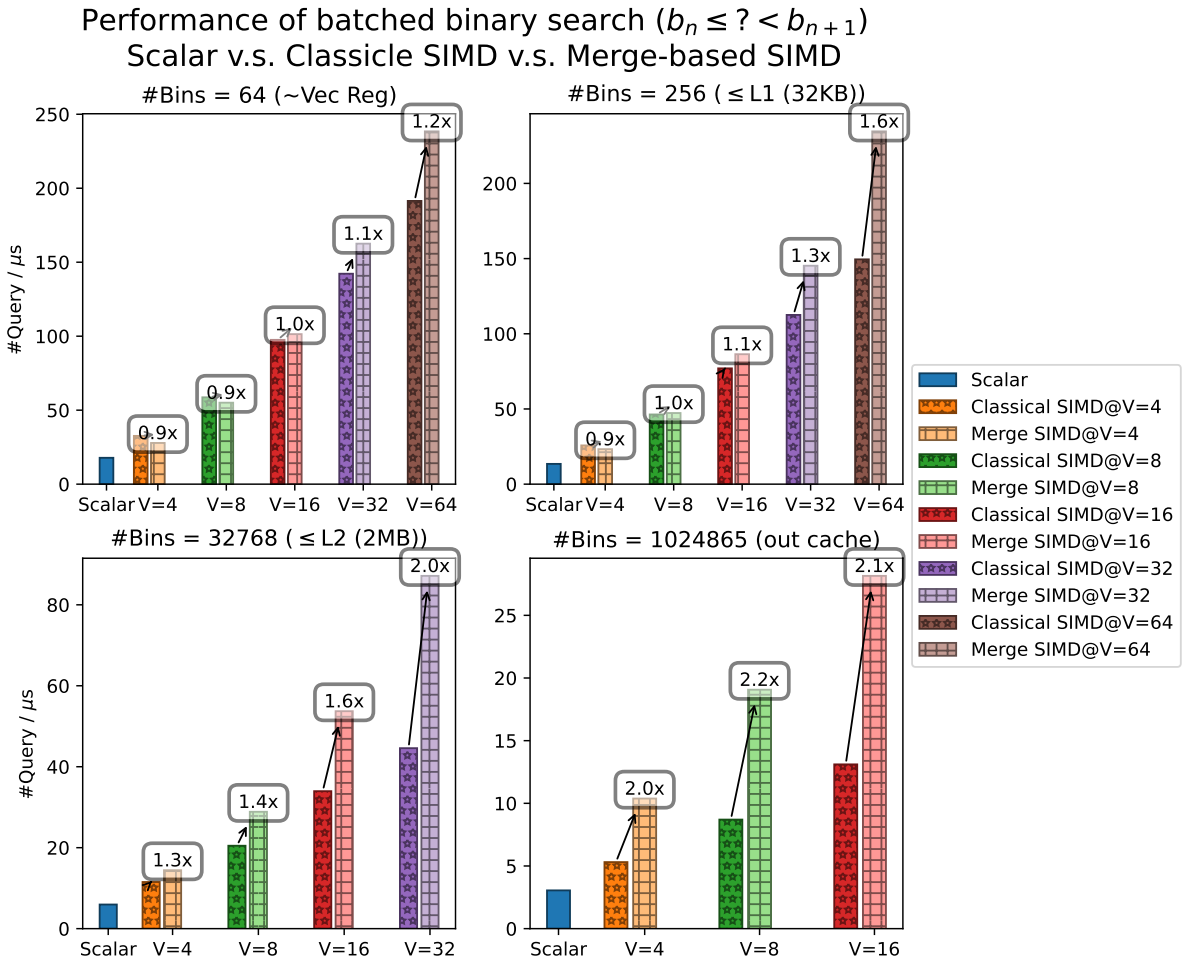


Figure 3.2: The performance of SIMD binary search using the new merge-SIMD primitives versus the classical SIMD implementation. While both implementations provide significant speedup over the scalar version, the new merge-based SIMD solution offers extra speedup when the number of bins is large and cannot fit into L1 or L2. This is due to the streaming access, which eliminates the need for SIMD gather.

The traditional way to parallelize binary search is to use each SIMD lane to execute an independent binary search, effectively behaving like V small cores running in lockstep. However, while the computation part can be accelerated by a factor of V, the memory load operation (a "gather" operation) to the middle elements does not improve with the

SIMD width V, becoming the bottleneck. This is because the load/store unit in modern processors can only handle one or two (constant) distinct addresses per cycle, while the memory load address diverges in different SIMD lanes in the binary search.

To overcome this bottleneck, we introduce a new algorithm that eliminates the "gather" memory load. Suppose the length of the delimiter array is S and the number of queries is Q, with a focus on the case $Q \geq S$. We divide the query array into segments of size near S, selecting $S' = 2^{\lceil \log_2 S \rceil}$ as a reasonable choice. This gives us $\lceil Q/S' \rceil$ segments in total. We process each segment in order, sorting the query keys of a segment together with their index in the original array as a payload. We then perform a two-way merge (with interpolation rule) of the sorted segment with the delimiter array, generating a result array consisting of matched pairs $(DIdx, QIdx)$ where $QIdx$ is the index of the query key in the original array and $DIdx$ is the index of the matched delimiter key. Finally, we scatter the $DIdx$ values to the result array using the $QIdx$ values as the scatter address. The overall algorithm is shown in Algorithm 4.

This modification brings two benefits without increasing the time complexity. First, all computation steps can be vectorized using SIMD. Specifically, step 2 and 3 (sorting and interpolation merge) can be implemented as a two-way merge operation and accelerated using the MSO SIMD primitives. Second, all memory load/store operations, except the last scatter, are vector accesses of V continuous elements. This eliminates the memory load ("gather") bottleneck and improves the throughput of the algorithm.

Note that although the last scatter step is still non-continuous, it is faster than the previous "gather"-based implementation for two reasons: 1) the scatter is invoked only once at the end, instead of $\log_2(S)$ times in a loop, and 2) the scatter is not on the critical path in the new algorithm; i.e. the core does not have to wait for it to finish. In contrast, the gathered data in the original algorithm has data dependency, making it part of the critical path.

Finally, the time complexity remains $O(Q \log(S)/V)$, as we have Q/V segments and the sorting and interpolation merge steps are $O(S \log S/V)$ and $O(S/V)$, respectively. This is the same as the original SIMD binary search algorithm $O(Q \log(S)/V)$.

3.3 K-way Additive Merge

In this section, we cover an operation called "K-way additive merge" and present an algorithm to effectively utilize MSO primitives for its implementation. Unlike the previous two sections, this operation requires careful analysis of the algorithm's time complexity, as we have discovered an input pattern that can lead to significant performance degradation if the algorithm is not designed thoughtfully. A robust new algorithm, along with a formal proof of its time complexity, is provided.

We organize this section as follows: First, we introduce the concept of K-way additive merge and explain why we cannot simply adopt the merge sort algorithm. Next, we present a new algorithm termed the α-merge based merge scheme, which is designed specifically for K-way additive merge. Finally, we analyze the time and space complexity of the α-merge based merge scheme.

3.3.1 Concept of K-way additive merge

The concept of K-way additive merge originated from operations in sparse linear algebra. It involves merging k sorted lists into a single sorted list while consolidating elements with identical keys found in all lists into a single element. When the elements in the list consist of pairs of keys and values, the lists can be thought of as dictionaries. The process can be generalized to include a reduce operator (often addition) that is applied to the values of identical keys.

K-way additive merge is often used to implement sparse accumulators where partial

sums in sparse multiplication operations form the sorted lists that need to be merged[1]. Table 3.1 shows several examples.

There are two classic ways to implement K-way additive merge: using a heap or iteratively performing two-way merges.

Heap-based method: One classic way to implement K-way additive merge uses a heap that tracks the smallest element in each list. The algorithm starts by adding the first element of each list to the heap. Then, the smallest element is removed from the heap and added to the result list, potentially consolidated with the last element already in the result list. The next element from the list the smallest element came from is added to the heap. This process is repeated until all elements are added to the result list. This algorithm has a time complexity of $O(n \log k)$, where n is the total number of elements in all lists, and k is the number of lists. A limitation of this algorithm is that it fails to exploit the consolidation of identical elements to reduce the number of comparisons.

Iterative merge: Another way to implement K-way merge is by performing a series of two-way merges. This algorithm is suitable for our MSO SIMD primitives, so we will focus on this method.

3.3.2 Why we cannot just imitate nature merge sort

The iterative K-way merge algorithm looks very similar to the nature merge sort, but simply borrowing the methods used in merge sort leads to severe problems. The purpose of this subsection is to explain the differences.

Suppose we have k sorted arrays $l_1, \ldots, l_k$ as input, and they have lengths $|l_1|, \ldots, |l_k|$, listed as follows:

$$l_1, l_2, \ldots, l_k$$

[1]This usually requires the operands to be stored in a sorted format, such as compressed sorted rows (CSR).

We can take arbitrary pairs of arrays (e.g., l_a and l_b) and merge them (additively) into a new array l'_{ab}, leading to $k-1$ remaining arrays to merge as follows:

$$l_1, \ldots, l_{a-1}, l_{a+1}, \ldots, l_{b-1}, l_{b+1}, \ldots, l_k; l'_{ab}$$

Finally, the above procedure is iterated $k-1$ times to obtain the last result array. The cost of merging two arrays l_a and l_b is proportional to the sum of their lengths $|l_a| + |l_b|$. Therefore, it is always preferable to merge short arrays with short arrays and long arrays with long arrays, and avoid merging a short array with a long array. Up to this point, the procedure is very similar to merge sort. To pick a good merge order to minimize cost, one may consider using the same merge order as in sorting, such as Huffman-tree-based merge order, or the TimSort algorithm. However, we found that the truth is the opposite. The Huffman-tree based merge order can have significantly worse space complexity, while the TimSort based algorithm can have significantly worse time complexity.

These problems ultimately stem from one subtle difference: in additive merge, if multiple elements have the same key, their value will be combined, so the result array will be shorter than the sum of the lengths of the input arrays.

$$|\text{Merge}_{\text{Sort}}(l_a, l_b)| = |l_a| + |l_b|$$
$$|\text{Merge}_{\text{Additive}}(l_a, l_b)| \leq |l_a| + |l_b|$$

Following is an example,

$$l_1.\text{keys} = \{1, 2, 3\}$$

$$l_2.\text{keys} = \{1, 3, 5\}$$

$$\text{Merge}_{\text{Additive}}(l_1, l_2).\text{keys} = \{1, 2, 3, 5\}$$

$$\text{Merge}_{\text{Sort}}(l_1, l_2).\text{keys} = \{1, 1, 2, 3, 3, 5\}$$

This difference has three consequences:

1. The dynamic nature of intermediate length in additive merge will introduce some noise to the scheduling of the merge order, which may lead to a suboptimal merge order.

2. Algorithms for additive K-way merge can aim for better time complexity because the total length reduces instead of being kept constant after every round of two-way merge.

3. The iterative version of this algorithm for additive K-way merge can aim for better space complexity ($O(|\bigcup_i l_i|)$, the length of the output array) instead of simply $O(\sum_i |l_i|)$ (the total length of all input arrays).

The problem with Huffman-tree based merge order

A method to schedule the merge order statically is to build a Huffman-tree using the length of each array as the weight of each leaf node. The non-leaf nodes represent intermediate result arrays. Then, an array's merge order is the path from the leaf node to the root node. However, the order obtained from Huffman-tree has two limitations:

First, because the Huffman-tree computes the weight of non-leaf nodes by summing the weights of its two children, this weight does not consider the possible consolidation of the same intermediate array and may not accurately reflect the actual length of the intermediate result array (it may be significantly larger than the actual length). This sum of two children provides misleading information for the selection of the merge order. Consider an extreme case where a subset of 2^n input arrays has exactly the same set of keys (say k keys), when they are merged together, the length of the result array will still be k while their weight in the Huffman-tree will be $2^n k$. In general, any static scheduling method can obtain a suboptimal merge order because the exact length of an additive two-way merge is not predictable and known in advance.

Second, the input arrays are dynamically generated in many application scenarios, such as sparse linear algebra. This means it is possible to accept one input array at a time and incrementally merge it with the existing intermediate arrays. Since additive merge can reduce the total size when identical keys are consolidated, the space complexity can be kept small. In contrast, the Huffman-tree merge order is not suitable for this case because it needs to wait for all input arrays to be available before a Huffman-tree can be built. Moreover, the shortest pair of two arrays may be located at the beginning and end of all arrays. Both of these factors force the Huffman-tree-based merge order to buffer the entire input arrays before beginning, which consistently demands a space complexity of $O(\sum_i |l_i|)$, where l_i is the length of the i-th input array. Ideally, the space complexity should be $O(|\bigcup_i l_i|)$, the length of the output array.

We should note that both limitations are related to the consolidation of identical keys that reduce the size of merged arrays. The Huffman-tree-based merge order is still good for merge sort. On the one hand, the length of the merged array (in merge sort) is fully predictable. On the other hand, there is no regret about the $O(\sum_i |l_i|)$ space complexity since there is no way to make it smaller anyway.

The problems with TimSort's stack-based merge order

The TimSort algorithm is the default sorting algorithm in Python and Java. It extends the idea of merging sorts and deciding the merge order of many existing sorted arrays using an elegant strategy that is cheap to evaluate. Moreover, there are already formal proofs of its time complexity stating that when merging k arrays $l_1, l_2, \ldots, l_k$, the worst-case time complexity is $O(\sum_i |l_i| \log k)$ (weaker version) or $O\left((\sum_i |l_i|) \cdot (H_l + 1)\right)$ (stronger version) where H_l is the entropy of lengths (i.e., $H_l = l'_i \log_2(l'_i)$, $l'_i = l_i / \sum_i l_i$).

However, when the same strategy is applied to additive merge, it degrades to significantly worse quadratic time $O((\sum_i |l_i|)^2)$ (which is $\leq O(\sum_i |l_i| \times k) \leq O(\sum_i |l_i| \log k)$) in the worst case. Even worse, this worst case is not rare; a simple set of randomly generated arrays can trigger this worst case. It can easily become 10x, 100x, 1000x (or in fact, arbitrary) times worse than a naive merge order in the shape of a full binary tree.

The catastrophic failure of TimSort's strategy is rather tricky, but it is ultimately caused by the reduced result length of merge results, which merge sort does not have, i.e., $Merge_{\text{Additive}}(l_1, l_2) \leq |l_1| + |l_2|$. With this background, we make two contributions:

1. We show that a one-line modification to TimSort's strategy is sufficient to overcome the problem. The modified strategy is still based on a stack and is called the α-merge based merge order.

2. We analyze the space and time complexity of the new strategy and obtain new upper bounds that are very different from sorting.

Since both TimSort's strategy and the α-merge based merge order are based on a stack, we will first introduce the α-merge strategy and then revisit how TimSort's strategy fails in this section.

3.3.3 The α-merge based merge scheme

The α-merge-based merge scheme is a method for dynamically scheduling merging orders and reducing total merge costs. It has been previously discussed in the context of sorting [20]. Although the algorithm looks similar to our additive merge, we have found that its time and space complexity are different (as we have already seen in the example of TimSort), and so we reanalyze them in this subsection.

The algorithm starts by initializing an empty stack and then processing each of the input arrays. The stack may contain intermediate arrays that are not yet merged, and it needs to maintain an invariant about the length of arrays in the stack. Let's denote the arrays in the stack S as $S = r_1, r_2, \ldots$, where r_1 is the bottom of the stack, and $|r_i|$ denotes their length. The number of arrays in a stack S is denoted as $[S]$. The invariant states that the stack needs to maintain $|r_{i+1}| \leq \alpha |r_i|$ for all i. The parameter α is a constant that controls the size of the intermediate arrays and is usually set to 0.618 (the inverse of the golden ratio).

To process an input array, the array is first added to the top of the stack. Then, the stack is checked to see if the invariant is violated. If it is, then the top two arrays ($r_{[S]}$ and $r_{[S]-1}$) or the second two arrays ($r_{[S]-1}$ and $r_{[S]-2}$) are merged, whichever is cheaper. This process is repeated until the invariant is satisfied. When all the input arrays are processed, the stack will be merged from the bottom to the top and squashed into one array. The entire procedure is shown in Algorithm 5.

In Algorithm 5, the function `FixStack` is first used to maintain the invariant (line 6) and then collapse the stack (line 8). It iteratively merges two of the top three arrays in the stack until the condition test returns false. In practical implementation, the condition test for all possible i (the "$\exists i$ such that $|s[i]| > \alpha |s[i-1]|$") can be simplified to test only $i = [S]$ and $i = [S] - 1$ (between the top three arrays), because at any moment, only the

current top two arrays can differ from the end of the last invocation of `FixStack` in line 6.

This α-merge-based method is suitable for additive merge because it adapts to the length of earlier merge results instead of being predetermined ahead of time.

3.3.4 Analysis of the α-merge based merge scheme

In this subsection, we present our results on the space and time complexity of the α-merge based merge scheme. The proofs for these results are provided in the Appendix at the end of this chapter.

Theorem 3.3.1. The peak memory footprint of the α-merge based merge scheme is asymptotically optimal and bounded by the number of unique keys $|Key|$ in all input arrays, i.e., $|Keys| = |\cup_{i=1}^{k} l_i|$.

$$\text{Peak Memory } \leq O(|Key|)$$

Theorem 3.3.2. (Weaker Version) The time complexity of the α-merge is following:

$$\text{Time Complexity } = O\left(I \cdot \left(H_l - \log\left(\frac{I}{X}\right) + 1\right)\right)$$

where

$$I = \sum_{i=1}^{k} |l_i| \text{ is the input size}$$

$$H_l = -\sum_{i=1}^{k} \frac{|l_i|}{I} \log\left(\frac{|l_i|}{I}\right) \text{ is the entropy computed using input lengths}$$

$$X = |\cup_{i=1}^{k} l_i| = |Keys| \text{ is the output size}$$

Notice that the term $O(I \cdot (H_l + 1))$ coincides with the lower-bound of any merge sort, which can be achieved using a Huffman-tree to decide the merge order. The rest of the term $-I \cdot \log(I/X)$ is obviously negative. Therefore, we can confirm that our k-way merge scheme runs at least as fast as the sorting problem.

If we look closer at the negative term, we can see that it shows how eliminating duplicate tuples helps to reduce the time complexity. The ratio I/X is the ratio of the input size to the output size, which indicates the degree of eliminated duplicates during the merge. The larger the ratio, the more time is saved. Consider two extreme cases:

1. If the keys in the input arrays are almost unique and there are few tuples with identical keys to be merged into one during the execution, the ratio I/X is close to 1.

2. If there are a lot of duplicated keys in the input arrays, so the output size X is much smaller than the input size I, and the ratio I/X can be a huge number.

Finally, it is important to note that $I \cdot (H_l - \log(I/X)) \leq I \cdot H_l \leq I \log k$, where k is the number of input arrays. Therefore, the α-merge-based scheme is at least not worse than trivially using a complete binary tree.

The upper bound of the time complexity can be strengthened by replacing the term X (the number of all distinct keys globally) with the average working set size $\overline{W}$, which represents the number of active keys at any given moment. The average working set size $\overline{W}$ can be much smaller than the number of all distinct keys globally X if the input arrays are not in a random order, and the keys appearing in one array are also likely to appear in recent input arrays before it.

Theorem 3.3.3. (Stronger Version) The time complexity of the α-merge is following:

$$\text{Time Complexity } = O\left(I \cdot \left(H_l - \log\left(\frac{I}{\overline{W}}\right) + 1\right)\right)$$

where

$I = \sum_{i=1}^{k} |l_i|$ is the input size

$H_l = -\sum_{i=1}^{k} \frac{|l_i|}{I} \log\left(\frac{|l_i|}{I}\right)$ is the entropy computed using input lengths

$\overline{W}$ is the average working set size, defined following:

Definition 3.3.1. The average working set size $\overline{W}$ is defined as the geometric mean of the per-element working set of each input element $W_{e,t}$:

$$\overline{W} = \exp\left(\frac{\sum_{i=t}^{k} \sum_{e \in l_t} \log(W_{e,t})}{\sum_{i=t}^{k} \sum_{e \in l_t} 1}\right)$$

Definition 3.3.2. The per-element working set size $W_{e,t}$ is defined as the number of distinct keys that appeared in the input array since the last time e appeared, plus the length of the current input array l_t.

$$W_{e,t} = \left| \bigcup_{i=last+1}^{t-1} l_i \right| + |l_t|$$

$$\text{where } last = \begin{cases} 0 & \nexists i < t, e \in l_i \\ \max\{i < t | e \in l_i\} & \text{otherwise} \end{cases}$$

Corollary 3.3.4. The time complexity of the α-merge is following:

$$\text{Time Complexity } = O\left(I \cdot (H_l + H_{Key} - \log(I) + 1)\right)$$

where

$$I = \sum_{i=1}^{k} |l_i| = \sum_{e \in Keys} |C_e| \text{ is the input size}$$

$$H_l = -\sum_{i=1}^{k} \frac{|l_i|}{I} \log\left(\frac{|l_i|}{I}\right) \text{ is the entropy computed using input lengths}$$

$$H_{Key} = -\sum_{e \in Keys} \frac{|C_e|}{I} \log\left(\frac{|C_e|}{I}\right) \text{ is the entropy computed using all distinct keys}$$

$$C_e = \{i | e \in l_i\} \text{ is the indexes of input arrays that contain key } e$$

This result becomes more elegant when considering the sparse patterns of all input arrays as a probability matrix. The upper bound can then be expressed as the mutual information between the rows and columns:

$$\begin{aligned}\text{Time Complexity } &= O\left(I \cdot (H(Y) + H(Z) - H(Y,Z) + 1)\right) \\ &= O\left(I \cdot (\mathcal{I}(Y;Z) + 1)\right)\end{aligned}$$

Specifically, we need to represent the input arrays as a sparse matrix, where each row corresponds to an input array and each column corresponds to a key. The number of rows is k, and the number of columns is $|Keys|$. This matrix can be seen as the joint probability distribution of a pair of random variables (Y, Z), where all non-zero elements have an equal probability of $1/nnz$, where $nnz = I = \sum_i l_i$. In this representation, $H_l = H(Y)$ is the entropy of random variable Y, and $H_{Key} = H(Z)$ is the entropy of

random variable Z. Finally, we have $\log(I) = \log(nnz) = H(Y, Z)$.

The results have interesting applications. When the mutual information $\mathcal{I}(X;Y)$ between the rows and columns is small, the time complexity is also small. This is the case in the following situations:

- When the number of input arrays is small (k is small) or the length of input arrays is highly unbalanced (H_l is small). This is because $\mathcal{I}(X;Y) \leq H(X) \leq \log(k)$.
- When the number of distinct keys is small ($|Keys|$ is small) or the number of frequencies of each key is highly unbalanced (H_{Key} is small). This is because $\mathcal{I}(X;Y) \leq H(Y) \leq \log(|Keys|)$.
- When the sparsity pattern of different input arrays is highly similar. This makes $H(X) + H(Y) \approx H(X, Y)$, therefore $\mathcal{I}(X;Y) \approx 0$.

The results presented in this section are highly related, with Theorem 3.3.3 being the strongest. The corollary 3.3.4 and the weaker version of time complexity are consequences of Theorem 3.3.3.

Lemma 3.3.5. The logarithm of the average working set size is upper-bounded by the entropy of keys:

$$\log \overline{W} \leq H_{\text{Keys}}$$

Proof. We first observe that the per-element working set of element e at the t-th input array covers the keys since the last time it appears in the input array, i.e., the input arrays with index $last + 1, last + 2, \ldots, t$. This is a disjoint range from the coverage of another occurrence of the same e on another t'-th input array with $t \neq t'$. We denote $C_e = t | e \in l_t$, then we have:

$$\forall e \in Keys, \quad \sum_{t \in C_e} W_{e,t} \leq \sum_{t=1}^{k} |l_t| = I$$

Therefore, we have:

$$\begin{aligned}
\sum_{t \in C_e} \log(W_{e,t}) &\leq |C_e| \log\left(\frac{\sum_{t \in C_e} W_{e,t}}{|C_e|}\right) \\
&\leq |C_e| \log\left(\frac{I}{|C_e|}\right) \\
&= -I \cdot \left(\frac{|C_e|}{I}\right) \log\left(\frac{I}{|C_e|}\right)
\end{aligned}$$

Also realize that iterating over keys and over input arrays are interchangeable:

$$\sum_{e \in Keys} \sum_{t \in C_e} \cdots = \sum_{t=1}^{k} \sum_{e \in l_t} \cdots$$

Therefore, we have:

$$\begin{aligned}
\log \overline{W} &= \frac{\sum_{i=t}^{k} \sum_{e \in l_t} \log(W_{e,t})}{\sum_{i=t}^{k} \sum_{e \in l_t} 1} \\
&= \frac{\sum_{e \in Keys} \sum_{t \in C_e} \log(W_{e,t})}{\sum_{e \in Keys} \sum_{t \in C_e} 1} \\
&\leq -\sum_{e \in Keys} \left(\frac{|C_e|}{I}\right) \log\left(\frac{I}{|C_e|}\right) \\
&= H_{Keys}
\end{aligned}$$

□

Therefore, the Theorem-3.3.3 implies Collary-3.3.4 and also Theorem-3.3.2 if we no-

tice that $H_Key \leq \log(|Key|) = \log(X)$:

$$\underbrace{I \cdot \left(H_l - \log\left(\frac{I}{\overline{W}}\right) + 1\right)}_{\text{bound in Theorem-3.3.3}} \leq \underbrace{I \cdot (H_l + H_{Key} - \log(I) + 1)}_{\text{bound in Collary-3.3.4}}$$
$$\leq \underbrace{I \cdot \left(H_l - \log\left(\frac{I}{X}\right) + 1\right)}_{\text{bound in Theorem-3.3.2}}$$

Therefore, we only need to prove the main Theorem-3.3.3, which is left to a separate Section-3.3.6 due to its length.

3.3.5 Appendix 1: Proof of the space complexity

The space complexity is related to the peak memory footprint (M) of the stack.

Case 1: The invariant hold on the stack; this is the status before pushing a new array into the stack at line-4.

$$\forall i, |stack[i]| \leq \alpha |stack[i+1]|$$

Suppose there are n arrays in the stack, then we have memory footprint M_{hold}:

$$M_{hold} = \sum_{i=0}^{n-1} |stack[i]| \leq |stack[n-1]| \times \sum_{i=0}^{n-1} \alpha^i$$
$$< |stack[n-1]| \times \frac{1}{1-\alpha}$$

Because intermediate arrays do not contain duplications, they must be shorter than

the total number of distinct elements in the input, which is $O(|\bigcup_i l_i|)$

$$|stack[n-1]| \leq |\bigcup_i l_i|$$

Therefore, we have

$$M_{hold} < \frac{|\bigcup_i l_i|}{1-\alpha}$$

Case 2: After pushing the input array but before invoking `FixStack` at line-6, the stack may violate the invariant. But because the stack satisfies the invariant before the push, its memory footprint $M_{postPush}$ is:

$$\begin{aligned} M_{postPush} &\leq M_{hold} + \max_i |l_i| \\ &< (\frac{1}{1-\alpha} + 1)|\bigcup_i l_i| \end{aligned}$$

Case 3: During invoking `FixStack` at line-6. Because `FixStack` iteratively merges two arrays at a time, while $\text{Merge}_{\text{Additive}}$ can only reduce or keep the size of the merged array compared to its two inputs, the memory footprint M_{Fix} strictly decreases. Therefore, we have

$$M_{Fix} \leq M_{postPush}$$

Case 4: During invoking `FixStack` at line-8, the total length of the arrays in the stack is strictly decreasing. Because the stack invariant hold when the function is called at line-8, we have:

$$M_{Collaps} \leq M_{hold}$$

In summary, the overall space complexity M is the maximum of the prior four cases:

$$M = \max(M_{hold}, M_{postPush}, M_{Fix}, M_{Collaps}) \leq O(|\bigcup_i l_i|)$$

The space complexity is $O(|\bigcup_i l_i|)$, which is asymptotically optimal because the size of the result array is $|\bigcup_i l_i|$. When the ratio of identical keys is high, this $O(|\bigcup_i l_i|)$ space complexity is much smaller than the $O(|\sum_i l_i|)$ space complexity of the original merge sort.

3.3.6 Proof of the time complexity

Notations:

We have k input arrays. They are $l_1, l_2, \ldots, l_k$, their length are denoted as $|l_1|, |l_2|, \ldots, |l_k|$. The total length of all input arrays is $\sum_{t=1}^{k} |l_t|$, the number of unique keys in the input arrays is denoted as $|Key| = |\cup_{i=1}^{k} l_k|$. Here we abuse the union notation to denote the number of unique keys from a set of arrays after duplication removed.

We denote the stack state before pushing the t-th array as S_t. The state immediately after pushing the t-th array is denoted as F_t^0. A sequence of merged will be performed. The state after the h-th merge after push the t-th array is denoted as F_t^h. The total number of merge triggered after push the t-th array and before the next push is m_t. The state after the final input array being processed is denoted as S_{k+1}. It is followed by the final collapse step. The state after the h-th merge after the final collapse is denoted as G^h and there are m_G merges in final collapse.

Therefore, the stack goes through following sequence of states: each row represent the statis of the stack after a push, and each column represent the state after a merge; the every new row represent the state after the next push.

$$
\begin{aligned}
&S_1 \to F_1^0 \to F_1^1 \to F_1^2 \to \cdots \to F_1^{m_1-1} \to \\
&S_2 \to F_2^0 \to F_2^1 \to F_2^2 \to \cdots \to F_2^{m_2-1} \to \\
&\cdots\cdots \\
&S_k \to F_k^0 \to F_k^1 \to F_k^2 \to \cdots \to F_k^{m_k-1} \to \\
&S_{k+1} \to G^0 \to G^1 \to G^2 \to \cdots \to G^{m_G}
\end{aligned}
$$

We donate the total merge cost of a sequence of state transition $X \to \cdots \to Y$ as $C(X \to \cdots \to Y)$. Obviousely, $C(X \to \cdots Y \to \cdots Z)$ is the sum of $C(X \to \cdots Y)$ and $C(Y \to \cdots Z)$. Therefore, the total cost of the whole merge process can be decomposed as following:

$$
\begin{aligned}
C(S_1 \to \cdots \to G^{m_G}) &= C(S_1 \to \cdots \to S_{k+1}) + C(S_{k+1} \to G^0 \to \cdots \to G^{m_G}) \\
&= \sum_{t=1}^{k} C(S_t \to F_t^0 \to \cdots \to F_t^{m_t-1} \to S_{t+1}) \\
&+ C(S_{k+1} \to G^0 \to \cdots \to G^{m_G})
\end{aligned}
$$

The first term ($\sum_{t=1}^{k} C(S_t \to F_t^0 \to \cdots \to F_t^{m_t-1} \to S_{t+1)}$) is the cost of pushing all the input arrays into the stack. The second term ($C(S_{k+1} \to G^0 \to \cdots \to G^{m_G})$) is the cost of final collapse. We investigate them seperately to get the upper bound of the total cost.

Definition 3.3.3. The last occurrence array index $L(e;t)$ of a key e against a prefix set of of t input arrays $L_t \coloneqq [l_1, l_2, \cdots, l_t]$ is defined as the index of the last input array

in L_t that contains e (if $t = 0$, $L_t = \emptyset$). If e does not appear in any array of L_t, then $L(e;t) = 0$.

$$L(e;t) = \begin{cases} 0 & \text{if } \nexists i, e \in l_i \\ \max\{i < t | e \in l_i\} & \text{otherwise} \end{cases}$$

Definition 3.3.4. The block reuse distance $R(e;t)$ of a key e against a prefix t input arrays $L_t \coloneqq [l_1, l_2, \cdots, l_t]$ is defined as the number of distinct keys that have been encountered since the array of the last occurrence of e in L_t.

$$R(e;t) = \left| \bigcup_{i=L(e;t)+1}^{t-1} l_i \right|$$

Proposition 3.3.6. The total merge cost is bounded as follows:

$$C(S_1 \to \cdots \to G^{m_G}) \leq O\left(\sum_{t=1}^{k} \sum_{e \in l_t} 1 + \log\left(1 + \frac{R(e;t)}{|l_t|}\right)\right)$$

Note this bound is identical to the one in Theorem-3.3.3 after some transform if only we notice that $R(e;t) + |l_t| = W_{e,t}$.

The cost of final collapse:

We first investigate the cost of final collapse.

Proposition 3.3.7. The cost of the final collapse is bounded by $O(|Key|)$.

$$C(S_{k+1} \to G^0 \to \cdots \to G^{m_G}) \leq O(|Key|)$$

Proof. We denote the number of arrays in stack before the final collapse (S_{k+1}) as $[S_{k+1}]$

, and laytout it from the bottom to the top as $r_1, r_2, \cdots, r_{[S_{k+1}]}$, as shown in Figure-3.3.

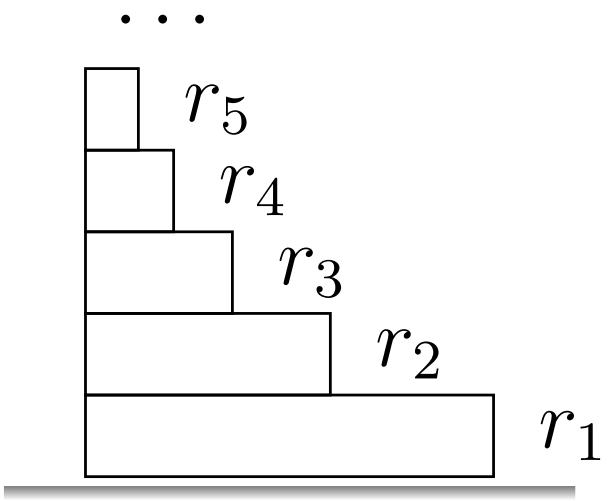


Figure 3.3: The stack before the final collapse, i.e. status S_{k+1}.

When the stack get collapsed into a single array in the final state (G^{m_G}) using a sequence of merges, the merge order can be represented as a binary tree, as shown in Figure-3.4. The root of the tree is the final array, and the leaves are the arrays in the stack S_{k+1}. The non-leaf node also represent the result of a merge, whose cost is bounded by the sum of the length of the two children. Therefore, the cost of the final collapse is bounded by the weighted sum of the length of leaves with the depth of the tree as weight. We denote d_i as the depth of the r_i in the tree.

$$C(S_{k+1} \to G^0 \to \cdots \to G^{m_G}) \leq \sum_{i=1}^{[S_{k+1}]} |r_i| \cdot d_i$$

Due the rule of merge operation, the status of stack must satisfy all the constraint about the length that the length of the array must decrease exponentially by a factor of α from the bottom to the top, therefore:

$$|r_i| \leq \alpha^{i-1}|r_1| \leq \alpha^{i-1}|Key|$$

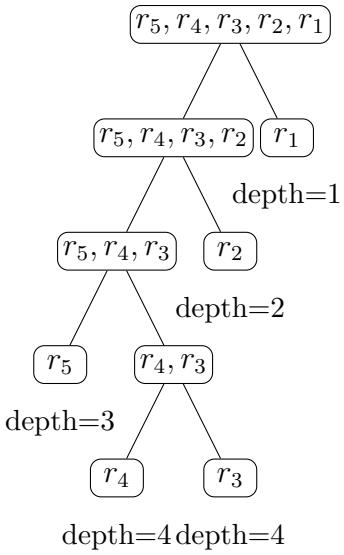


Figure 3.4: Binary tree rooted at ABCDE with new node names and rounded rectangle nodes.

Meanwhile, the merge rule only considerd merge the two top arrays, or the second top array and the third top array. Therefore, the i-th array r_i will not participate merge until the remaining size of the stack is less than $i + 2$. Afterward, even the array participate every subsequent merge, the total number of merge it participate will be less then the number of remaining array substract one. Therefore, the depth of the i-th array r_i is bounded by $i + 1$:

$$d_i \leq i + 1$$

Therefore, the total merge cost is bounded as following:

$$
\begin{aligned}
C(S_{k+1} \to G^0 \to \cdots \to G^{m_G}) &\leq \sum_{i=1}^{[S_{k+1}]} |r_i| \cdot d_i \\
&\leq \sum_{i=1}^{[S_{k+1}]} \alpha^{i-1} |Key| \cdot (i+1) \\
&\leq |Key| \times \left(\sum_{i=1}^{\infty} \alpha^{i-1} \cdot (i+1) \right) \\
&\leq |Key| \times \frac{2-\alpha}{(\alpha-1)^2} \\
&\leq O(|Key|)
\end{aligned}
$$

The cost of pushing all the input arrays into the stack

We now focus on the merge cost of the first term ($\sum_{t=1}^{k} C(S_t \to F_t^0 \to \cdots \to F_t^{m_t-1} \to S_{t+1)}$). We denote the status of the stack after pushing the i-th input array as S_t. We denote the number of arrays in stack after pushing the i-th input array as $[S_t]$.

First recall that there are to cases when performing a merge on the stack: merge the top two arrays, or merge the second top array and the third top array. We denote them as Type A and Type B respectively, as illustrated in Figure-3.5.

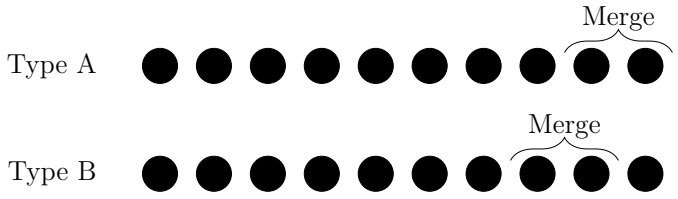


Figure 3.5: The two types of merge.

From the state S_t to S_{t+1}, there are totally m_t merges. We can divide them into two halves by taking the all contiguous Type B merge in the front as the first halves, and all the rest as second half. We denote the number of merge of the first group as p. For

example, we may list the m_t merges as below:

$$\overbrace{\underbrace{B\ B\ \cdots\ B}_{(p \text{ leading Type B merges})}\quad\underbrace{A\ A\ B\ \cdots}_{\substack{(m_t-p) \text{ rest merges of Type A or B,}\\ \text{lead by } A\\ \text{Or this part can be empty as a whole.}}}}^{m_t \text{merges}}$$

Therefore, we decompose the cost of pushing an input array and the merges afterwards into two parts, with the first part start from state S_t and end at state F_t^p, containing the first p merges, and the second part start from state F_t^p and end at state S_{t+1}, containing the rest $m_t - p$ merges.

$$C(S_t \to F_t^0 \to \cdots \to F_t^{m_t-1} \to S_{t+1)}) = \underbrace{C(S_t \to F_t^0 \to \cdots \to F_t^p)}_{\text{cost of first } p \text{ merge}} + \underbrace{C(F_t^p \to \cdots \to F_t^{m_t-1} \to S_{t+1)})}_{\text{cost of rest } m_t - p \text{ merge}}$$

Proposition 3.3.8. The cost of the first p merges after pushing the t-th input array is bounded by the length of input array $|l_t|$ by a constant time.

$$C(S_t \to F_t^0 \to \cdots \to F_t^p) \leq O(|l_t|)$$

Proof. We denote the array in the stack after pushing the t-th input array as $r_1, r_2, \cdots, r_{[S_t]}$ which is similar to the final collapse step shown in Figure-3.3. Then F_t^0 is simply appending the t-th input array to the end of the stack, which is equivalent to $r_1, r_2, \cdots, r_{[S_t]}, l_t$. In the following subsequent p merges of Type B, the second and third top array remaining in the stack will be merged. So the state $S_t, F_t^0, F_t^1, \cdots, F_t^p$ can be illustrated as following:

$$S_t = r_1, r_2, \cdots, r_{[S_t]}$$

$$F_t^0 = r_1, r_2, \cdots, r_{[S_t]}, l_t$$

$$F_t^1 = r_1, r_2, \cdots, r_{[S_t]-2}, Merge(r_{[S_t]-1}, r_{[S_t]}), l_t$$

$$F_t^2 = r_1, r_2, \cdots, r_{[S_t]-3}, Merge(r_{[S_t]-2}, Merge(r_{[S_t]-1}, r_{[S_t]})), l_t$$

$$\cdots$$

$$F_t^p = r_1, r_2, \cdots, r_{[S_t]-p-1}, Merge(r_{[S_t]-p}, Merge(r_{[S_t]-p+2}, \cdots, Merge(r_{[S_t]-1}, r_{[S_t]}))), l_t$$

Therefore, the total merge cost from S_t to F_t^p is bounded the weighted sum of the length of the arrays in the stack, as the content in $r_{[S_t]-p+0}, r_{[S_t]-p+1}, \cdots, r_{[S_t]-1}, r_{[S_t]}$ will participate no more than $1, 2, \cdots, p-1, p, p$ times in the merge ($r_{[S_t]}$ is a special case).

$$C(S_t \to F_t^0 \to \cdots \to F_t^p) \leq \sum_{i=0}^{p} |r_{[S_t]-p+i}| \cdot (i+1)$$

Then, because S_t is in a state no merge can be triggered, so their length satisfy the exponential descreasing property:

$$|r_{i+1}| \leq \alpha |r_i|, \quad \text{for all } i \in 0, \cdots [S_t] - 1$$

In addition, because Type B are triggered, to satisfy the condition, the top array (l_t) must be longer than the third top array in states $F_t^0, F_t^1, \cdots, F_t^{p-1}$, therefore we have:

$$|l_t| \geq |r_{[S_t]-p}|$$

In summary, we have:

$$\begin{aligned} C(S_t \to F_t^0 \to \cdots \to F_t^p) &\leq \sum_{i=0}^{p} |r_{[S_t]-p+i}| \cdot (i+1) \\ &\leq |r_{[S_t]-P}| \sum_{i=0}^{p} \alpha^i \cdot (i+1) \\ &\leq |l_t| \frac{1}{(1-\alpha)^2} \\ &\leq O(|l_t|) \end{aligned}$$

□

Definition 3.3.5. The height function $h(e; S)$ of an key e on a stack S of arrays $S = (r_1, r_2, \cdots, r_{[S]})$ are defined as:

$$h(e; S) = \begin{cases} 0, & \nexists i, e \in r_i \\ \max\{i | e \in r_i\}, & \text{otherwise} \end{cases}$$

Definition 3.3.6. The height function $H(S)$ of a stack S of arrays $S = (r_1, r_2, \cdots, r_{[S]})$ are defined as:

$$H(S) = \sum_{e \in Keys} h(e; S)$$

where $Keys$ is the set of all unique keys in the input arrays $\cup_{i=1}^{[S]} l_i$. But since $h(e; S)$ is 0 by default if e is not in any array in the stack, so it can also use $\cup_{i=1}^{[S]} r_i$.

Proposition 3.3.9. For any subarray from sequence of state transition $F_t^p \to \cdots \to F_t^{m_t-1} \to S_{t+1}$, say $X \to Y \to Z \to \cdots \to W$, the merge cost $C(X \to W)$ is bounded by the decrease height function $H(S)$ of the stack S in state W by a constant time.

$$C(X \to Y) \leq O(H(X) - H(Y))$$

Proof. It is suffice to prove two adjecent states $X \to Y$ such that Y is the result of a merge operation after X.

Since $X \to Y$ undergoes either a Type A or Type B merge, we only focus on the top three arrays on the stack, and we denote them as U, V and W (from bottom to top) respectively.Therefore, we can represent X and Y as following:

$$X = \cdots, U, V, W$$

$$\text{Type A: } Y = \cdots, U, Merge(V, W)$$

$$\text{Type B: } Y = \cdots, Merge(U, V), W$$

Case 1: Y is the result of a merge operation of Type B. The merge cost of Type B is $C(X \to Y) = |U| + |V|$. In order to have Type B merge triggered, we have $|U| < |W|$. Because all distinct keys $e \in V \cup W$ has their height decreased by 1, while all other keys has their height unchanged, so we have:

$$\begin{cases} h(e; X) = h(e; Y) \text{ for all } e \notin V \cup W \\ h(e; X) = h(e; Y) + 1 \text{ for all } e \in V \cup W \end{cases}$$

Therefore, we have:

$$H(X) = H(Y) + |V \cup W|$$

$$\begin{aligned} C(X \to Y) &\leq |U| + |V| \\ &\leq |W| + |V| \\ &\leq 2|V \cup W| \\ &\leq 2(H(X) - H(Y)) \end{aligned}$$

Case 2: Y is the result of a merge operation of Type A. The merge cost of Type A is $C(X \to Y) = |V| + |W|$. In order to have Type A merge triggered, we have $|U| \geq |W|$. Because all distinct keys $e \in W$ has their height decreased by 1, while all other keys has their height unchanged, so we have:

$$H(X) = H(Y) + |W|$$

Now we only need to prove that $|V| \leq z \cdot |W|$ for some constant z. Realize that current stack status at X is the result of a sequence of Type A or Type B merge from the state F_t^0 (so X is F_t^v for some v), we can write down where different arrays in X are merged from in F_t^0:

$$\begin{aligned} &S_t = r_1, r_2, \cdots, r_{[S_t]} \\ &F_t^0 = r_1, r_2, \cdots, r_{[S_t]-1}, l_t \\ &X = F_t^v = r_1, r_2, \cdots, \underbrace{r_{[S_t]-v-1}}_{U}, \underbrace{Merge(r_{[S_t]-v}, \cdots, r_{[S_t]-v+w})}_{V}, \underbrace{Merge(r_{[S_t]-v+w+1}, \cdots l_t)}_{W} \end{aligned}$$

Because S_t is at the state no merge can be triggered, so $\alpha \cdot |r_i| > |r_{i+1}|$. Therefore, we

have a upper bound on $|V|$:

$$\begin{aligned}
|V| &\leq \sum_{i=0}^{w} \cdot |r_{[S_t]-v+i}| \\
&\leq |r_{[S_t]-v-1}| \cdot \sum_{i=0}^{w} \cdot \alpha^{i+1} \\
&\leq |U| \cdot \frac{\alpha}{1-\alpha} \\
&\leq |W| \cdot \frac{\alpha}{1-\alpha}
\end{aligned}$$

With the upper bound on $|V|$, we have:

$$\begin{aligned}
C(X \to Y) &\leq |V| + |W| \\
&\leq |W| \cdot \frac{\alpha}{1-\alpha} + |W| \\
&\leq |W| \cdot \frac{1}{1-\alpha} \\
&\leq \frac{1}{1-\alpha} \cdot (H(X) - H(Y))
\end{aligned}$$

Putting the two cases together, we have:

$$\begin{aligned}
C(X \to Y) &\leq \max(2, \frac{1}{1-\alpha}) \cdot (H(X) - H(Y)) \\
&\leq O(H(X) - H(Y))
\end{aligned}$$

□

Therefore, applying Proposition-3.3.9 to the merge cost of the whole sequence of state transition $F_t^p \to \cdots \to F_t^{m_t-1} \to S_{t+1}$, we have:

$$C(F_t^p \to \cdots \to F_t^{m_t-1} \to S_{t+1}) \leq O(H(F_t^p) - H(S_{t+1}))$$

Proposition 3.3.10. The change of height function from $H(S_t)$ to $H(F_t^p)$ can be bounded by the block reused distance of the input array being pushed into the stack with following relation:

$$H(F_t^p) - H(S_t) \leq O\left(\sum_{e \in l_t} 1 + \log\left(1 + \frac{R(e;t)}{|l_t|}\right)\right)$$

Proof. From F_t^0 to F_t^p, all merge operations are Type B merge, so the layout at F_t^p can be written as two parts: l_t and the rest (we denote as F_t^p/l_t)

$$F_t^p = \underbrace{r_1, r_2, \cdots, r_{[S_t]-p-1}, \overbrace{Merge(r_{[S_t]-p}, \cdots, Merge(r_{[S_t]-1}, r_{[S_t]}))}^{\text{Denote as } r'_{[S_t]-p}}}_{\text{part 1: } F_t^p/l_t}, \underbrace{l_t}_{\text{part 2:} l_t}$$

Because all arrays in F_t^p/l_t are merged from existing arrays in S_t and their height is decreased or unchanged, so we have:

$$H(S_t) \geq H(F_t^p/l_t)$$

Now we focus on $H(F_t^p) - H(F_t^p/l_t) = \sum_{e \in l_t} h(e; F_t^p) - h(e; F_t^p/l_t)$. For each $e \in l_t$, it appear on the top array l_t of F_t^p, so we have:

$$h(e; F_t^p) = [F_t^p] = [S_t] + 1 - p$$

In addition, $0 \leq h(e; F_t^p/l_t) \leq [F_t^p/l_t]$, we analysis case by case.

1. $h(e; F_t^p/l_t) = [F_t^p/l_t] - n$, for $n \geq 2$, then e is in an array $r_{[S_t]-p-n}$, which is below $r_{[S_t]-p-1}$. The array above $r_{[S_t]-p-n}$, i.e. $r_{[S_t]-p-n+1}$ are array untouched from S_t, and have

$$\alpha^{n-2} \cdot |r_{[S_t]-p-n+1}| \geq |r_{[S_t]-p-1}|$$

Because p is the number of the longest prefix of Type B merge sequence, the next merge, if it exists, must be $TypeA$ merge, and the merge rule demands that the third top array must be longer than the top array, i.e.

$$|r_{[S_t]-p-1}| \geq |l_t|$$

Because array $r_{[S_t]-p-n}$ contains the last input array that contain e before the t-th input array, therefore the array between them $r_{[S_t]-p-1}$ is the merged result of the subset of input array $l_{\text{Last}(e;t-1)+1}, \ldots, l_{t-1}$. Therefore,

$$R(e,t) \geq |r_{[S_t]-p-n+1}|$$

Putting everything together, we have for all $e \in l_t$,

$$\alpha^{n-2} R(e,t) \geq \alpha^{n-2} \cdot |r_{[S_t]-p-n+1}| \geq |r_{[S_t]-p-1}| \geq |l_t|$$

And we have (note $1/\alpha > 1$):

$$H(F_t^p) - H(F_t^p/l_t) = n \leq 2 + \log_{1/\alpha}\left(\frac{R(e,t)}{|l_t|}\right)$$

Note that we must have $R(e,t) \geq |l_t|$.

2. $h(e; F_t^p/l_t) = [F_t^p/l_t] - n$ for $n = 0, 1$, then e is part of the second or third top array in F_t^p/l_t, i.e. $r'_{[S_t]-p}$ or $r_{[S_t]-p-1}$. We directly have:

$$h(e; F_t^p) - h(e; F_t^p/l_t) \leq 2$$

Putting the two cases together, we have:

$$\begin{aligned} h(e;F_t^p) - h(e;F_t^p/l_t) &= \max(2, 2 + \log_{1/\alpha}\left(\frac{R(e,t)}{|l_t|}\right)) \\ &\leq 2 \cdot \left(1 + \log_{1/\alpha}\left(1 + \frac{R(e;t)}{|l_t|}\right)\right) \end{aligned}$$

And therefore:

$$\begin{aligned} H(F_t^p) - H(S_t) &= (H(F_t^p) - H(F_t^p/l_t)) - (S_t - H(F_t^p/l_t)) \\ &\leq H(F_t^p) - H(F_t^p/l_t) \\ &= \sum_{e \in l_t} h(e;F_t^p) - h(e;F_t^p/l_t) \\ &\leq O\left(\sum_{e \in l_t} 1 + \log\left(1 + \frac{R(e;t)}{|l_t|}\right)\right) \end{aligned}$$

□

Now consider the state transition from S_t to S_{t+1}, using Proposition-3.3.9 and Proposition-3.3.10

$$H(S_{t+1}) - H(S_t) = \underbrace{H(F_t^p) - H(S_t)}_{\leq O\left(\sum_{e \in l_t} 1 + \log\left(1 + \frac{R(e;t)}{|l_t|}\right)\right)} + \underbrace{H(S_{t+1}) - H(F_t^p)}_{\leq -C(F_t^p \to \cdots \to S_{t+1})}$$

We have:

$$C(F_t^p \to \cdots \to S_{t+1}) \leq O\left(\sum_{e \in l_t} 1 + \log\left(1 + \frac{R(e;t)}{|l_t|}\right)\right) + (H(S_t) - H(S_{t+1}))$$

Therefore, we can bound the total cost after pushing t-th input array and before the

$t+1$-th merge:

$$\begin{aligned} C(S_t \to F_t^0 \to \cdots \to F_t^{m_t-1} \to S_{t+1}) &= C(S_t \to \cdots \to F_t^p) + C(F_t^p \to \cdots \to S_{t+1}) \\ &\leq \underbrace{O(|l_t|)}_{\text{First half, by Proposition-3.3.8}} \\ &+ \underbrace{\leq O\left(\sum_{e \in l_t} 1 + \log\left(1 + \frac{R(e;t)}{|l_t|}\right)\right) + (H(S_t) - H(S_{t+1}))}_{\text{Second half}} \end{aligned}$$

Now we have all the necessary tools to prove the main theorem.

$$\begin{aligned} C(S_1 \to \cdots \to G^{m_G}) &= \sum_{t=1}^{k} C(S_t \to F_t^0 \to \cdots \to F_t^{m_t-1} \to S_{t+1}) \\ &+ C(S_{k+1} \to G^0 \to \cdots \to G^{m_G}) \\ &\leq O\left(\sum_{t=1}^{k} |l_t|\right) \\ &+ O\left(\sum_{t=1}^{k} \sum_{e \in l_t} 1 + \log\left(1 + \frac{R(e;t)}{|l_t|}\right)\right) \\ &+ H(S_0) - H(G^{m_G}) \\ &+ \underbrace{O\left(|Keys|\right)}_{\text{By proposition-3.3.7}} \\ &\leq O\left(\sum_{t=1}^{k} \sum_{e \in l_t} 1 + \log\left(1 + \frac{R(e;t)}{|l_t|}\right)\right) \end{aligned}$$

Here we use the fact that $H(S_0) = 0$ for the initially empty stack, $H(G^{m_G}) \geq 0$ for any stack, and $|Keys| = |\cup_{t=1}^{k} l_t| \leq \sum_{t=1}^{k} |l_t|$.

□

3.3.7 Appendix 2: The problem with the TimSort scheme

We need to exercise caution when applying sorting algorithms to additive merge, despite their apparent similarity. The noise caused by the reduced result array length can be misleading for the scheduling algorithm and significantly impair performance. One example is the TimSort scheme, which is similar to the α-merge-based merge except for using a different condition test to determine when to collapse the stack. The condition test is "$\exists i$ such that $|s[i+1]| \geq |s[i]|$ or $|s[i+1]| + |s[i+2]| \geq |s[i]|$." In sorting, the TimSort scheme is nearly identical to the α-merge scheme, with α set to the golden ratio, 0.618. However, in additive merge, the TimSort scheme can take quadratic time in the worst case to the input size. An example that would cause TimSort to be extremely slow is the following:

$$
\begin{aligned}
l_1.\text{keys} &= [1, 2, 3, \ldots, n] \\
l_2.\text{keys} &= [1, 2, 3, \ldots, n-1] \\
l_3.\text{keys} &= [1] \\
l_4.\text{keys} &= [1] \\
&\ldots \\
l_n.\text{keys} &= [1]
\end{aligned}
$$

Using the TimSort scheme, the stack will contain l_1 and l_2 from bottom to top. Then, starting from $i = 3$, each time a new array l_i is pushed into the stack, the task looks like

this:

$$
\begin{aligned}
&----\text{bottom}----\\
&r_{[S]-2} = [1, 2, 3, \ldots, n]\\
&r_{[S]-1} = [1, 2, 3, \ldots, n-1]\\
&r_{[S]} = [1]\\
&----\text{top}----
\end{aligned}
$$

Because $|r_{[S]}| + |r_{[S]-1}| \geq |r_{[S]-2}|$, a merge will be triggered between $r_{[S]}$ and $r_{[S]-1}$. The result array will still be $[1, 2, 3, \ldots, n-1]$, and the stack will restore to the following:

$$
\begin{aligned}
&----\text{bottom}----\\
&r_{[S]-1} = [1, 2, 3, \ldots, n]\\
&r_{[S]} = [1, 2, 3, \ldots, n-1]\\
&----\text{top}----
\end{aligned}
$$

This means that for all $3 \leq i \leq n$, there will be a very inefficient merge between $[1]$ and $[1, 2, 3, \ldots, n-1]$, resulting in n merge costs. Therefore, the total merge cost is $O(n^2)$, scaling quadratically with the input size.

Unfortunately, the situation described above (which I call the "performance trap") is not a rare case that only happens in deliberately constructed scenarios; it can happen almost anywhere. The key to triggering this problem is that r_1 and r_2 contain almost all different keys, and r_1 is only slightly shorter than r_2. Then, for any new array (even

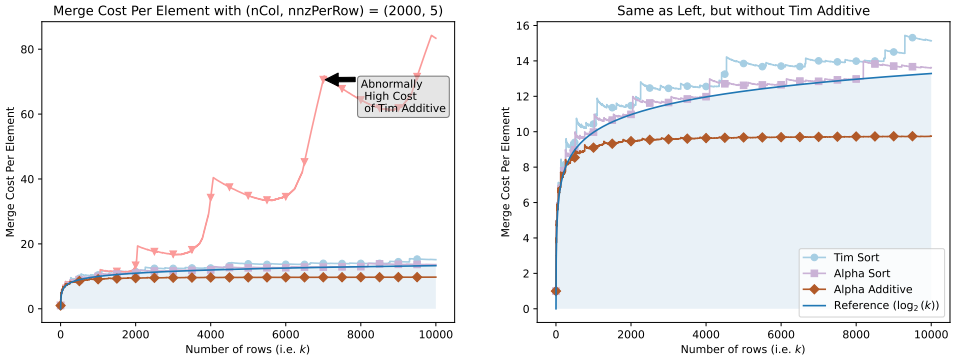


Figure 3.6: Abnormally high cost for additive merge when using a TimSort-like scheme on randomly generated input with an even distribution. One would naturally expect that any method performing well in merge sort should perform better in additive merge. However, this is not true for the TimSort-like scheme. On the other hand, the Alpha merge does not have this problem.

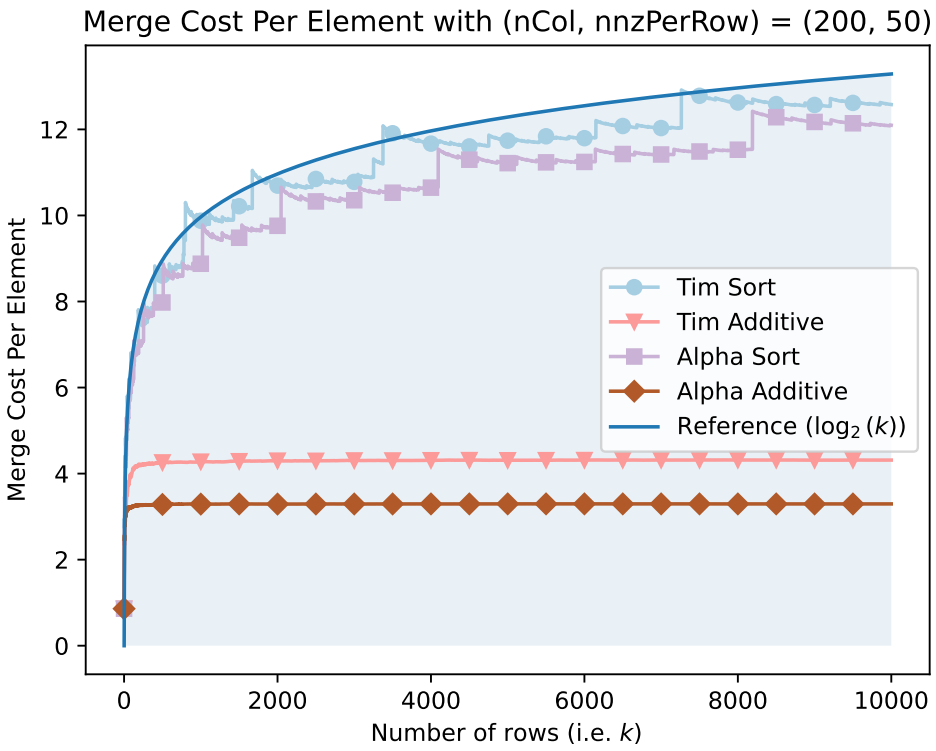


Figure 3.7: Case in which the TimSort-like scheme performs normally for additive merge.

a very short one) that is pushed to the stack, it will be immediately merged, while the new r_1 (the merged result) is still likely to be shorter than r_2, since most of the keys are already contained in the original r_1. This situation persists (whereas in merge sort, it would be escaped). For example, a set of randomly generated samples from a fixed set can trigger this problem, as shown in Figure 3.6. Only when the sparse array is very dense can the TimSort scheme escape from this situation and perform similarly to the α-merge scheme, as shown in Figure 3.7.

The ultimate cause of this problem is that the TimSort scheme is designed for sorting, where the result array is the same size as the input array. Hence, the above-inefficient merge of a length-1 and length $n-1$ array will happen only once instead of repeatedly.

Algorithm 4: SIMD-friendly Batched Binary Search

```
Input: delimiter array D of size S, query array Q of size Q.
Output: result array R of size Q.
begin
    S' ← 2^⌈log2 S⌉;
    D' ← zip(D, 0 ... S' − 1);
    D'[S'] ← (+∞, S');
    for i ← 0 to ⌈Q/S'⌉ do
        /* 1. Take the i-th segment of Q                          */
        for j ← 0 to S' do
            seg[j] ← (Q[i * S' + j], i * S' + j);
        end
        /* 2. Sort the segment by the query keys                  */
        Sort seg[0..S' − 1] by the first field;
        /* 3. Merge the segment with D using the interpolation rule */
        merged ← MergeInterpolate(D', seg[0..S' − 1], S');
        /* 4. Scatter the result to R                             */
        for j ← 0 to S' do
            R[merged[j].second] ← merged[j].first;
        end
    end
end
Function MergeInterpolate(d, q, lq) is
    pa, pb ← 0, 0;
    while pb ≤ lq do
        (keyA, valA) ← d[pa];
        (keyB, valB) ← q[pb];
        if keyA ≤ keyB then
            pa ← pa + 1;
        end
        else
            M[pb] ← (valA, valB);
            pb ← pb + 1;
        end
    end
    return M
end
```

Name	Expression	Relation
SpMSpV	$y_j = \sum_i A_{ij}x_i$ where A and x are sparse	$A[i,:]x_i$ forms sorted lists
SpMSpM	$C_ij = \sum_k A_{ik}B_{jk}$ where A and B are sparse	Method 1: $A[i,k]B[k,:]$ forms sorted lists Method 2: $A[:,k]B[k,:]$ forms sorted lists
SpMSpT	$C_ijl = \sum_k A_{ik}T_{kjl}$ where A and T are sparse	$A[i,k]T[k,:,:]$ forms sorted list

Table 3.1: Sparse multiplication operations are implemented using k-way additive merge

Algorithm 5: The α-merge based merge scheme

Input: input arrays $l_1, l_2, \ldots, l_k$, constant α
Output: merged array m

```
// Following, if s is a stack, s[i] denotes the i-th array from the bottom.
1  begin
2      stack ← ∅;
3      for i ← 1 to k do
4          stack.push(l_i);
5          FixStack(stack, lambda (s) {
6              ∃i such that |s[i+1]| > α|s[i]| });
7      end
8      FixStack(stack, lambda (s): s.size >= 2);
9      return stack[0];
10 end
11 Function FixStack(stk, condTest) is
12     while stk.size ≥ 2 and condTest(stk) do
           // d is the depth of the stack.  p1, p2, p3 are the top three arrays in the
              stack.
13         d ← [stk];
14         if stk.size = 2 or |stk[d]| ≤ |stk[d − 2]| then
15             p1, p2 ← stk.pop(), stk.pop() ;
16             p'1 ← Merge_Additive(p1, p2);
17             stk.push(p'1);
18         else
19             p1, p2, p3 ← stk.pop(), stk.pop(), stk.pop();
20             p'2 ← Merge_Additive(p2, p3);
21             stk.push(p'2), stk.push(p1);
22         end
23     end
24 end
```

Chapter 4

Reduce-by-key Operation on Unordered (Key,Value) Operation

In this chapter, we present a method for performing a reduce-by-key operation on an unsorted (key, value) array. The new method offers several advantages over classic methods, such as hash tables or sorting, in terms of parallelism and memory access traffic/pattern. This makes the method more suitable for modern architectures with massive parallelism on a single chip, such as GPUs, and layered memory hierarchies.

In prior research, this operator is typically studied as part of a complete application, where reduce-by-key is used in pairs with another upstream operator that produces the (key, value) pairs, forming a frontend-backend structure. Frontends in different applications are typically different, while the backend is the same. Although prior work have different intended application domains, they often focus on attacking the same challenges in the backend. On the other hand, the application-dependent frontend usually turns out to be simpler and less central. In this work, instead of focusing on one specific application, we fully concentrate on the backend part and leave the frontend optimization problem to future full system-level integrators.

Why do we need yet another method? For a small-scale problem and when using a single thread, a textbook hash table suffices. However, if the number of threads scales to thousands (such as an Nvidia A100 GPU or an accelerator) and the problem size scales to gigabytes (such as a large genomic dataset), hash tables and their variants are limited by inefficient random memory access and write-conflict. Other methods, such as using sorting or partition preprocessing, can alleviate the problem but lead to other problems, such as increased memory traffic, increased peak memory footprint, or reliance on prior knowledge about the input. None of those methods is optimal in all aspects (we summarized a checklist of features that applications from different domains may need, see Table-4.1).

We observe that earlier methods can be broadly categorized into two mindsets: "lazy deduplication" and "eager deduplication," which represent two opposite extremes. The advantages and disadvantages of these methods arise from the selection of one of these mindsets, with the opposite advantages and disadvantages resulting from selecting the other. In our approach, we employ an "incremental deduplication" scheme at every layer of the memory hierarchy to achieve the best of both worlds. Additionally, our method includes a memory-friendly deduplication method that stores all partial results in SDHA arrays, which serve as dedicated containers.

Numerical evaluation using simulations has shown that the new method offers advantages on parallel hardware over existing methods, including hash, sorting, and their variants or hybrids. However, these advantages are presented more directly by theoretical analysis, which establishes a relation between the method's memory footprint, traffic, access pattern, and that of the theoretical optimal of any design.

4.1 Applications with a Frontend/Backend Structure

In recent years, emerging applications like graph processing, genome analysis, sparse computing, and database have attracted researchers' interest in developing domain-specific accelerators. Many of them can be formulated in a "frontend-backend" structure logically.

- Frontend: The frontend generates a stream of tuples in a form of (key, val).
- Backend: Recieve the tuple stream from the frontend and aggregate tuples with identical keys using a reduce operator (usually +, max, or min).

Following are some examples.

Graph computing: Graph computation is widely used in social networks, machine learning and other problems. Many graph algorithms can be implemented by iteratively pushing the information from an active subset of vertices to their neighbor as updates, usually known as the "push-style" method (Sample-Code 1). It will iterate through the active vertices and generate an update for every neighbor vertices. Next, all updates to the same vertex will be aggregated. The second phase of this problem can be seen as a reduce-by-key operation. Each update can be seen as a tuple where the key is the id of the neighbor vertex. The challenge of this graph workload is how to resolve the write-conflict and random memory access when aggregating those updates[21, 22, 23].

```
Input: graph (V,E), active vertices V_A ⊆ V,
vertex data D_v, edge data D_E
Output:  updated vertices data F_v
Customizable:  function op, reduce operator Reduce
Initialize F_v into a empty dictionary
for u in V_A:
    for v in neighbor_E(u):
        F_v[v] = Reduce(F_v[v], op(D_v[v], D_E[(u,v)]))
```

Sample Code 1: Push-style method for graph computation

Genome Analysis: With the development of the DNA sequencing technique, an estimation of zettabytes of data has been collected and which greatly benefits the areas like Physiology or Medicine, especially during the Covid-19 pandemic. K-mer counting is one of the most frequently used tool to discover the relevance between DNA sequences. The algorithm will scan the whole DNA sequence with a sliding window size of K and count the number of occurrences for each k-length subsequence (k-mer). Such a counting problem can be seen as a "reduce-by-key operation" by treating each occurrence of a k-mer as a tuple. The performance of K-mer counting not only suffers from random access and write-conflict but also a huge dictionary that can easily become tens or even hundreds of GB due to the huge number of distinct k-mers in a large DNA dataset [24, 25, 26, 27, 28, 29]. The latter imposes a high standard of space efficiency and any implementation that trades space for speed, such as using data replication, will be not affordable. The algorithm is shown in Sample-Code 2.

```
Input: a DNA sequence
Output:  a dictionary dict for the counting
Parameter:  window size k
Initialize dict into an empty dictionary
for i in range(len(DNA)-K):
    seed = DNA[i:i+k]
    dict[seed] += 1
```

Sample Code 2: K-mer counting

Sparse Linear Algbra The multiplication between sparse vectors, matrices, or tensors with a reduced dimension can be seen as a "reduce-by-key" operation. Taking the general sparse matrix multiplication (SpGEMM) for example, the sparse partial sums can be seen as the tuples to aggregate by seeing the row and column indices in the result matrix as keys[30, 31, 32, 33, 34, 35, 36, 37, 38]. Sparse computing requires the "reduce-by-key" operation to be performed in parallel with the partial sums generation, i.e. to be

on-line. This is because the number of intermediate partial sums is usually huge and off-line algorithms may run out of memory even if the size of the result matrix is small. Therefore they require an on-line algorithm to aggregate tuples at the same speed as the multipliers that generate those tuples, instead of dumping all partial sum tuples to memory and performing an off-line aggregation in the end. For example, a sort-based SpGEMM (off-line) implementation CUSP[31] and bhSparse[34], despite being faster, run out of memory more frequently than the hash table-based methods[38].

Consider how a well-known example, the outer-product implementation of general sparse matrix-matrix multiplication SpGEMM, can be written in the form of reduce-by-key operation as shown in Sample-Code 3.

```
Input: sparse matrices A (CSC) and B (CSR)
Output:  sparse matrix C = AB
Initialize dict C into an empty dictionary
for i in range(k):
    for (row_idx, va) in A.sparseColumn(i):
        for (col_idx, vb) in B.sparseRow(i):
            C[(row_idx,col_idx)] += va * vb1
```

Sample Code 3: Outer-product implementation of SpGEMM

Database: Group-by aggregation is one of the most frequently used operators in databases. It is a "reduce-by-key operation" by seeing each row as a tuple[39, 40, 41, 42, 43, 44]. In databases, the data often have highly skewed distributions that are non-uniform both in frequency (e.g. power-law) and temporal location (e.g. moving cluster). Such non-uniformness can severely degrade the parallel computation performance by amplifying the contention's propobility. This requires an skew-resilient implementation to obtain a robust performance across different input distributions.

Table 4.1: Features supported by different methods (left) and demanded by applications (right) ($|Input|$: size of the input. $|Key|$: number of unique keys in the input. P: number of parallel threads.)

	Feature	Supported by methods					Required by applications			
		Sorting	Shared Hash	Private Hash	Partition+Hash	Proposed (SDHA)	Graph	Genmoic	Sparse	Database
Parallelism	Write-Conflict-Free	✓	✗	✓	✓	✓	Yes	Yes	Yes	Yes
	Skew-Resilience	✓	✗	✓	✗	✓	Yes			Yes
Memory Access	Random-Access-Free	✓	✗	✗	✓	✓	Yes	Yes	Yes	Yes
	Reduced traffic	✗	✓	✓	✗	✓	Yes	Yes	Yes	Yes
Space	On-line	✗	✓	✓	✗	✓			Yes	
	Near-optimal storage (Optimal is |Key|)	✗$O(\|Input\|)$	✓$O(\|Key\|)$	✗$O(P \times \|Key\|)$	✗$O(\|Input\|)$	✓$O(\|Key\|)$		Yes		

4.1.1 The checklist

Different application domains impose different subset technical requirements on the "reduce-by-key" operation. We summarize a checklist shown in the right side of Table-4.1. They can be categorized into three aspects:

- Parallelism
  - Write-conflict-free: It should effectively reduce the overhead of write conflict.
  - (+) Skew-resilience: It should maintain robust performance even for highly skewed input. Since skewness amplifies the probability of write-conflict, it is a strictly higher requirement than write-conflict-free.
- Memory access
  - Random-access-free: It should not have fine-grained random access to off-chip memory.
  - Minimum traffic: It should eagerly aggregate tuples on-chip to minimize the total data traffic to off-chip memory. For example, if the capacity of any level of cache is big enough to hold all distinct keys, zero traffic will spill out of that cache.
- Space

- On-line: It should not demand all tuples be generated before starting.
- (+) Near-optimal storage: The memory footprint should not be more than the necessary amount by a constant factor; specifically, the footprint should not depend on both input length and the number of threads. It is a strictly higher requirement than being on-line.

We marked (+) above for two entries (skew-resilience and near-optimal storage), since the other entry above them is the prerequisite for them: a skew-resilient algorithm must be write-conflict-free, and a near-optimal storage algorithm must be on-line.

4.1.2 The strength and limitation existing methods

We summarize the situation with existing methods on the left side of Table-4.1.

Sort-based method: The sort-based method first sorts all of the input tuples according to the key, followed by a linear scan to merge identical keys. Sorting algorithms such as merge sort have good parallelism, are resilient to skewed distribution, and have no random access, but they need to wait for all tuples to be ready (therefore off-line) and thus require large storage that is proportional to $|input|$. In addition, sorting usually has larger data traffic than hash-table-based methods, especially when the number of distinct keys, i.e. the "key cardinality" $|Key|$ is smaller than the on-chip memory size.

Hash table (common): Hash table-based methods directly aggregate incoming tuples with an existing record. This reduced the memory footprint and also off-chip traffic if the hash table is used with a cache. However, hash probes are random memory access.

Shared hash table: The shared hash table method uses only a global hash table shared by all threads, which makes the memory footprint minimum. However, sharing forces it to design a safety mechanism to prevent write-conflict break the data structure. Sharing also makes the algorithm sensitive to the skewness of the input stream because skewness

can concentrate all parallel accesses into one spot.

Private hash table: Private hash table use duplicated tables for each thread, which eliminates write conflict. However, this significantly increases the memory footprint by the factor of parallelism P. It also indirectly causes more off-chip traffic because tuples on-chip with the same key will not be aggregated if they are owned by different threads – the privatization reduces the effective capacity of on-chip memory.

Partition + hash table: This method uses a preprocessing step to divide the input tuples into disjoint groups according to the keys, similar to a step in the radix sort. By assigning different groups into different threads, write conflict is prevented. However, the pre-processing step makes it an off-line algorithm and increases its memory footprint and traffic.

4.2 Duplication is Evil and Necessary Evil

The reduce-by-key operation can be thought of as a sequence of insertions into a record dictionary. For each new input tuple, the algorithm must determine whether to immediately search for a tuple with the same key in the dictionary and merge the two tuples, or to store the new tuple in a designated location and wait for a more opportune moment to merge it with others. The latter option allows for duplication of the same key in memory, which is not entirely unreasonable. Algorithms like partition-based hash tables preserve duplication in their first phase with the goal of reducing the range of random memory access and creating opportunities to parallelize write-conflicts, which can introduce several issues. However, the situation is not so straightforward. If duplication is maintained, it will interact with hardware efficiency in numerous ways, creating many issues while also being beneficial in overcoming other problems. To illustrate this point, consider two extremes:

Not allow any duplication:

All tuples with an identical key must be merged immediately when seen from the input array. An example is hash table-based algorithms.

Peak memory footprint: The memory footprint is minimized because no redundant tuples are stored. In particular, it is bounded by $O(|Key|)$, where $|Key|$ is the number of distinct keys.

Random access: Random access is inevitable not only because of the random projection of the "hash function" in a hash table but also in any algorithm where only duplication is strictly prohibited. In such algorithms, the access pattern must be at least as random as the key distribution since there is no other place to store tuples with the same key except the last tuple sharing the same key. However, random access can severely impact performance when the range exceeds the cache size.

Write conflict: When multiple threads run concurrently and duplication is not allowed, write conflict is inevitable because all threads must write to the same memory location for the same key.

Memory traffic: Performing deduplication eagerly helps reduce memory traffic by decreasing the total number of tuples that need to be moved in subsequent operations. Moreover, most hardware features a multi-layer memory hierarchy, and merging tuples at higher levels, such as the cache, can reduce traffic to lower levels, such as the main

Allow arbitrary duplication:

Merging tuples with identical keys can be delayed until the algorithm deems it appropriate. Examples include sorting-based algorithms and partition-based hash algorithms.

Peak memory footprint: The memory footprint is unbounded due to the delayed merge. It can be as large as the input array, $O(|input|)$.

Random access: Long-range random access can be avoided by writing the tuple to the end of an array out of a few arrays instead of immediately merging it with an existing one. After performing a few passes of rearrangement (like sorting and partition) to concentrate tuples with the same key in memory, subsequent operations enjoy better locality and can be more efficient.

Write conflict: When multiple threads run concurrently, write conflict can be avoided by allowing each thread to have its own copy of the tuple with the same key. The duplication can be merged in the end as a post-processing step.

Memory traffic: Delaying deduplication misses the opportunity to reduce traffic, even a "half-delay" can result in a serious penalty in memory traffic. For instance, if we are using parallel hardware with thousands of threads, such as a GPU, with a last-level cache that can hold approximately 1 million tuples. When we use eager deduplication,

memory. Depending on the number of distinct keys buffered in the cache, the higher ratio of traffic can be filtered out by ensuring zero duplication in the cache.

we can achieve almost zero traffic to memory when the number of distinct keys is less than 1 million, $|Key| < 1M$. However, when we use private hash tables, this threshold is reduced to 1000 keys. It's worth noting that even though the private hash table is already very "eager" in deduplication as it removes all duplication within each thread, it still suffers from delayed deduplication for tuples with the same key but from different threads.

The summary, the situation appears to be a dilemma:

- Eagerly deduplication is necessary: It keeps memory footprint low and memory traffic low, especially with a memory hierarchy.
- Eagerly deduplication can be expensive: Large-range random access and write-conflict cause performance degradation.

One solution is to first attempt eager deduplication with limited effort; if it turns out not beneficial or too expensive, fall back to non-eager algorithms. For example, Muller et.al.[45] first check the input tuple in a small hash table (which can fit into the cache) to see if it can be merged with an existing tuple. If not, proceed like a normal partition-based hash method. The method makes sense both for scenarios where the key cardinality ($|Key|$, number of distinct keys) is so small that all tuples can be absorbed by the cache resident hash table, and for scenarios where the key cardinality is so large that the cache resident hash table is ineffective. The algorithm automatically falls back to the non-eager algorithm (i.e., partition-based hash method), sacrificing memory footprint to trade for performance.

It is impossible to contain and not contain duplication in a piece of memory at the same moment, but we don't have to. The requirements from the above dilemma focus on

two different moments; thus we can design a data structure that breathes between the two states and answers a seemly impossible "yes and no" to the dilemma. We noticed that:

1. To avoid long-range random access and write-conflict, the key is to permit preserving duplication when inserting a new tuple to shared or off-chip memory.

2. To avoid unbounded memory footprint and high memory traffic, the key is to eliminate duplication 1) before spilling traffic from one memory layer to the next memory layer, or 2) when the total memory footprint exceeds the minimum necessary size by a few times.

We tweaked a new algorithm that uses incremental deduplication that, for the first time, allows us to achieve the best of both worlds by switching between duplication-rich and duplication-clean states. The overhead of repeated invoked deduplication steps, of course, should be minimized. This is achieved by reducing their frequency using a control strategy and by reducing each execution cost using a structure called "sorted and deduplicated hash array (SDHA)".

4.3 Algorithm

In this section, we introduce the algorithm used in our design. Due to its complexity, we will first present the sequential version of the algorithm in Subsection 4.3.1 and then discuss the parallel version in Subsection 4.3.4. Both versions rely on two building blocks, introduced in Subsection 4.3.2 and Subsection 4.3.3, respectively.

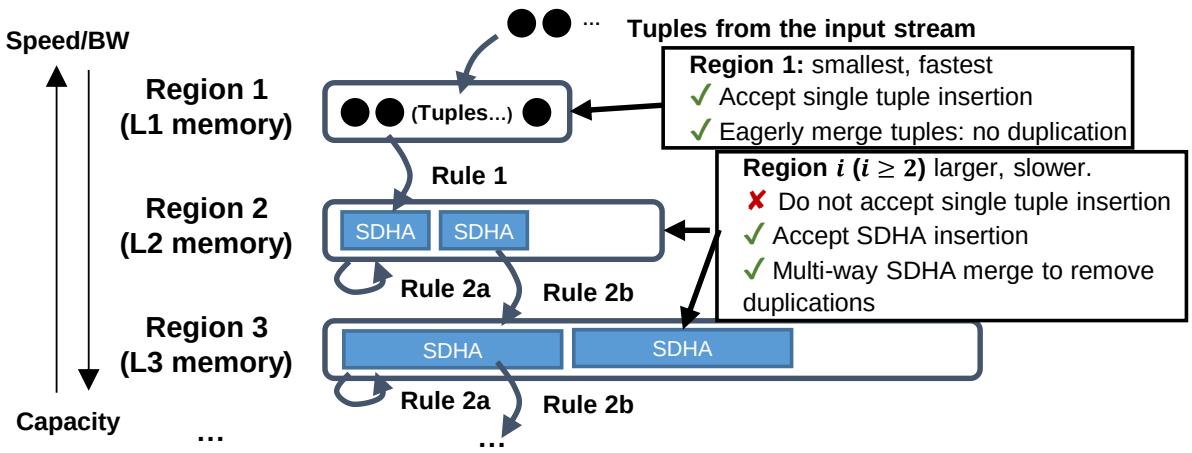


Figure 4.1: The organization of the SDHA data structure into the hierarchical memory.

4.3.1 A sketch of the SDHA build&merge algorithm

The algorithm will divide the storage space into a sequence of regions with exponentially growing sizes with a large multiply factor (ideally $64 \sim 128$) as shown in Figure-4.1. The insertion is made fast in amortize by ensuring that most insertions will only touch the smallest region, while larger regions will be touched exponentially less frequently. Meanwhile, the total memory footprint is kept low by managing the duplications in the largest region, which in turn set an upper bound for the total size of all regions, seeing it as the sum of a power series. When we map the algorithm into hardware, it will be best to map different regions to different levels of the memory hierarchy, for example, mapping region-1,2,3 and 4 to private scratch-pad, cluster-local scratch-pad, global shared scratch-pad, and DRAM, etc.

The smallest region supports single tuple insertion and immediately eliminates duplicated tuples. (Rule 1:) When it is full, it will dump all its tuples into the next region as a whole and in the form of a "sorted & deduplicated hash array (SDHA)". An SDHA is an array of tuples that 1) do not contain tuples with identical keys (aka "de-duplicated")

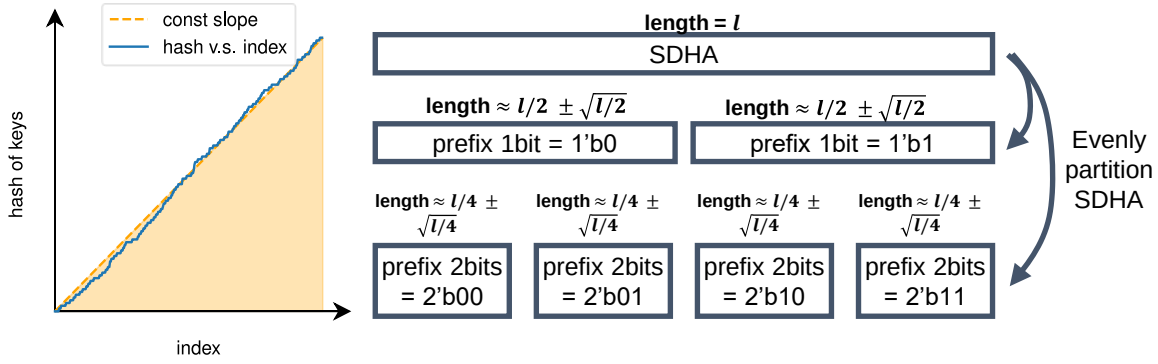
and 2) are sorted according to the hash value of their keys. After this, the smallest region will restore empty.

Starting from the second smallest region, single tuple insertion is NOT supported. Instead, they accept the insertion of a batch of tuples that are packed in an SDHA array at a time, which is received from their prior (smaller) region. The inserted SDHA array will be simply appended to the region. (Rule 2a:) When the region is nearly full such that won't be able to accept the insertion of the next SDHA array, all existing SDHA arrays in this region will be merged into a single SDHA array, which will remove all duplications between those arrays and create some extra empty space in this region. (Rule 2b:) But if the region is still more than 2/3 full after this step, the resulting SDHA array will be dumped to the next region, making this region completely empty.

The final rule to add is only applied to the largest non-empty region: (Rule 3:) instead of waiting for this particular region to be almost full to trigger merge as required by Rule 2a, we do it earlier when the total size of all SDHA arrays in this region is the larger than longest SDHA array in the region by a factor of $(1 + c)$, where the parameter c can take a small integer that have $c \geq 1$, such as 1 or 2. Note again that we don't apply this rule to other regions except the largest non-empty one.

The whole algorithm runs in a loop that each time process a tuple and applies the rules above. At the end of the algorithm, an epilogue phase will collapse the remaining SDHA into a single SDHA array, starting from the smallest region all the way down to the largest non-empty region, which will be presented as the final result of this algorithm. This epilogue will usually be short relative to the main loop because its time is only related to the number of unique keys ($|Key|$) instead of the input length ($|Input|$); we will see why later.

Because for each tuple, the hash value of its key will be used multiple times through the algorithm, we will store the hash value within the tuple to avoid re-computing. To



a) The hash values grow at nearly constant rate relative to their index in an SDHA array.

b) Using the p prefix bits of the hash values can nearly equally divide the SDHA array into 2^p segments. The deviation relative to the length of each segment is decreasing due the law of big numbers.

Figure 4.2: The uniformness of SDHA. The hash vaules are nearly linear to indices in the SDHA array.

avoid extra storage overhead, we will strip the key field from the tuple and replace it with its hash value. If we use an invertible hash function, the mapping between keys and their hash values is bijective, and we can recover the key from its hash value at the end of the epilogue.

The described algorithm so far looks very similar to the merge sort with early aggregation of tuples implemented during the merge phase, which can be seen as an optimization of the space and traffic. However, it will cause a much stronger and more profound impact when it is combined with SDHA's second requirement that it shall be sorted according to the hash value of keys instead of just keys — the uniformness. The uniformness, informally speaking, states that if we draw a plot of the hash values in an SDHA array against its index in the array, it looks very close to a straight line with a constant slope (Figure-4.2-a). This is a wonderful chemical reaction when the three keywords of SDHA meet together: sort, deduplicated, and hash. First, given a batch of input tuples, hashing will uniformly project different keys into the hash domain; while deduplication will wash off the difference in the frequency of keys; finally, sorting makes it monotone. The

uniformness is significant because it brings the following opportunities:

- Building an SDHA array from raw input tuples can be finished in linear time. (Used in Rule 1)
- Multi-way merging SDHA arrays with similar lengths can be finished in one pass in linear time. (Used in Rule 2a, 3)
- Segmenting an SDHA array using prefix bits of hash value will obtain subarrays with balanced length. (Shown in Figure-4.2-b. It is used in workload partition in the parallel version)

In the rest of this section, we will introduce the implementation of building an SDHA array in the smallest region for Rule 1 in Subsection-4.3.2. We discuss multi-way merging in Subsection-4.3.3. We discuss the parallel version of this algorithm in Subsection-4.3.4.

4.3.2 Building SDHA in linear time

Here we introduce a sub-algorithm that supports two operations: 1) the insertion of a tuple, 2) dumping all tuples as an SDHA array. The sub-algorithm is only used in the smallest region for Rule 1, therefore it will be mapped to the first level memory hierarchy which can be assumed to be the smallest but also the fastest.

It's known that some algorithms like Bucket Sort can sort an array with uniform distribution in linear time. Therefore, a modified version of Bucket Sort that enforces early aggregation of tuples with identical keys will satisfy our purpose (which also creates the uniformness required by the Bucket Sort itself). This modification makes the sub-algorithm look like a hash table as well. Another change to the standard bucket sort is to adjust the order of tuples in each bucket to make more recently used keys appear first in the bucket.

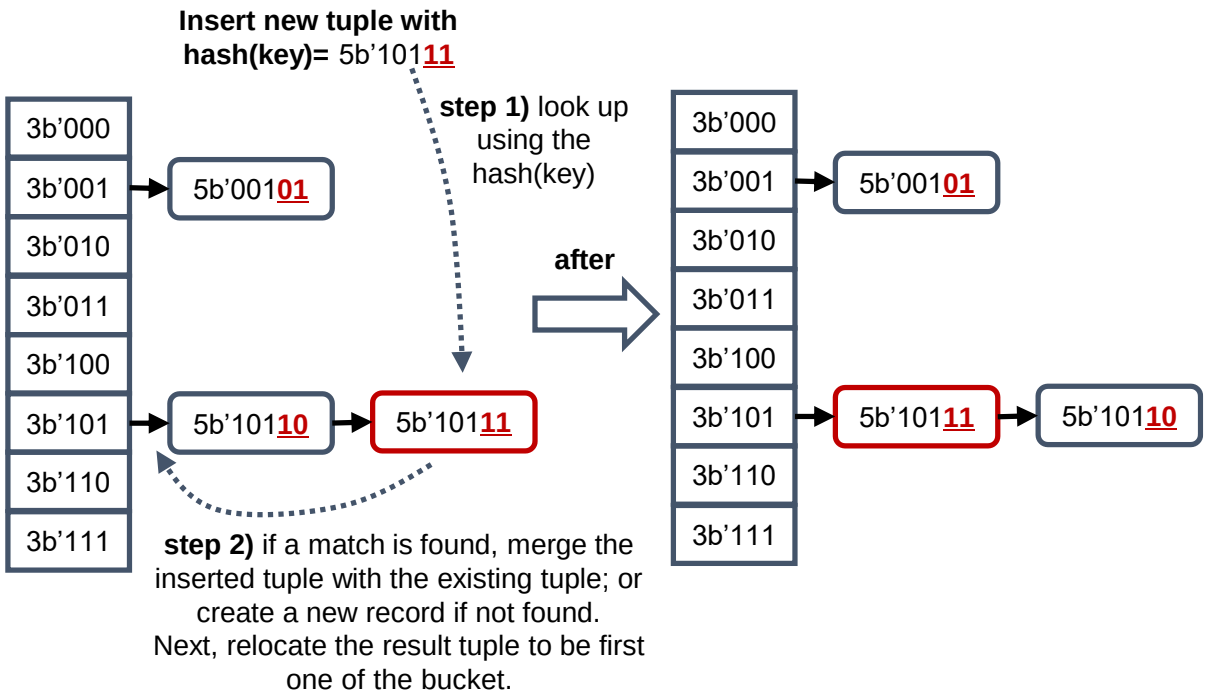


Figure 4.3: The insertion phase to build an SDHA in linear time: very similar to Bucket Sort and Hash Table, but the difference is that: 1) it uses the prefix bits of hash to determine the bucket, not the module of hash; 2) it has extra reordering after insertion; 3) the table size is intended to be very small to fit into a small SRAM and the insertion logic will be hardware accelerated — so it only solves small problems. Problems with larger key cardinality ($|Key|$) will be indirectly supported with the multi-way merge discussed latter.

The sub-algorithm begins with an empty table of buckets with 2^p entries where the p is selected to fit the capacity of the smallest region. The buckets are maintained using link lists. To insert a tuple, use the procedure defined in Algorithm-6.

When the smallness region is full, we will dump all content in the tuple as an SDHA array. To obtain a sorted (by hash) array, we dump the buckets in order. Most buckets will contain no more than 1 tuple. Some buckets may contain more than 1 tuple and those buckets need to be sorted internally. But because the probability to have more than one tuple in a bucket is exponentially decreasing with that number, a simple quadratic-time method is sufficient, which each time dumps the smallest tuple out of the bucket and

repeats until the bucket becomes empty. This procedure is defined in Algorithm-7. To accelerate the dumping operation, the hardware will implement a two-level bitmap to skip empty buckets.

Algorithm 6: Insertion a tuple in SDHA building

```
Input: A incoming tuple to be inserted
begin
    Compute the hash value of the key using an invertible hash function;
    Store the hash value in the tuple as a new field for later reuse while stripping the key field to save space;
    Use the p prefix bits of the hash value to index the table and get the head of the link list for the bucket;
    if A tuple with an identical hash(thus also key) is found in the link list then
        Aggregate the incoming tuple with the founded tuple;
        Move the tuple to the head of the link list;
    else
        Put the incoming tuple to the head of the link list;
    end
end
```

Algorithm 7: Dump the SDHA array

```
Output: An output SDHA array
begin
    for i ← 0, 1, 2, ..., 2^p − 1 do
        while bucket i is non-empty do
            Find the tuple with the smallest hash value in the bucket and append it to the end of the output
              SDHA array;
            Remove that tuple from the link list;
        end
    end
end
```

A natural question is: why do we restrict the usage of this sub-algorithm to only build short SDHA arrays in the smallest region and rely on multi-way SDHA merge at large scales? Or more directly, why don't we apply this mixed Bucket Sort&Hash Table directly at the largest scale? First, the latency of random access at a larger range is higher. Second and more importantly, we can cheaply duplicate a huge number of hardware units that hardcoded this sub-algorithm such that each of them has a small piece of private SRAM to be used for the Bucket Sort. Given a fixed resource budget, instantiating many smaller tables is better than instantiating a single large table — as the advantage of more parallelism can easily offset the disadvantage of requiring to perform the extra multi-way merge for larger problems, especially when the multi-way merge can

be finished in linear time.

4.3.3 Multi-way SDHA merging in linear time

We introduce a sub-algorithm that merges multiple SDHA arrays into one in linear time. It is used in all regions starting from the 2-th region for Rule 2a and Rule 3. We assume we have k SDHA arrays to merge and their lengths are $L_1, L_2, \ldots, L_k$ respectively. In addition, we normalize the size of a tuple to 1 (the unit size) and that the capacity of the i-th region is c_i.

If we simply treat those SDHA arrays as ordinary sorted arrays, then their multi-way merging is not a new problem. One common way is repeating the two-way merge operation, and another is finishing it in a single pass using a heap. Both methods are not linear time methods because they require $O(\log_2 k \times \sum_L l_i)$ time. However, we can improve it to linear time because SDHA arrays have uniformness.

Consider for a given integer q, we can use the q-bit prefix bits of the hash value to segment each SDHA array into 2^q segments. If we lay out those segments in a 2D matrix, we will see k rows and 2^q columns. Then, each column can be seen as a k-way merge problem with shorter input arrays; and we obtained 2^q subproblems in total as shown in Figure-4.4. If the subproblem is small enough, it can be directly handled by the mixed Bucket Sort&Hash Table introduced in Subsection-4.3.2.

We define the linear-time k-way merge algorithm recursively with simple devide-and-conquer relation:

- If k-way merge is triggered at the i-th region ($i \geq 3$), we divide the problem into 2^q subproblems to make them small enough to be solvable as a k-way merge problem by the $(i-1)$-th region.

- If k-way merge is triggered at the 2nd region, we divide the problem into 2^q sub-

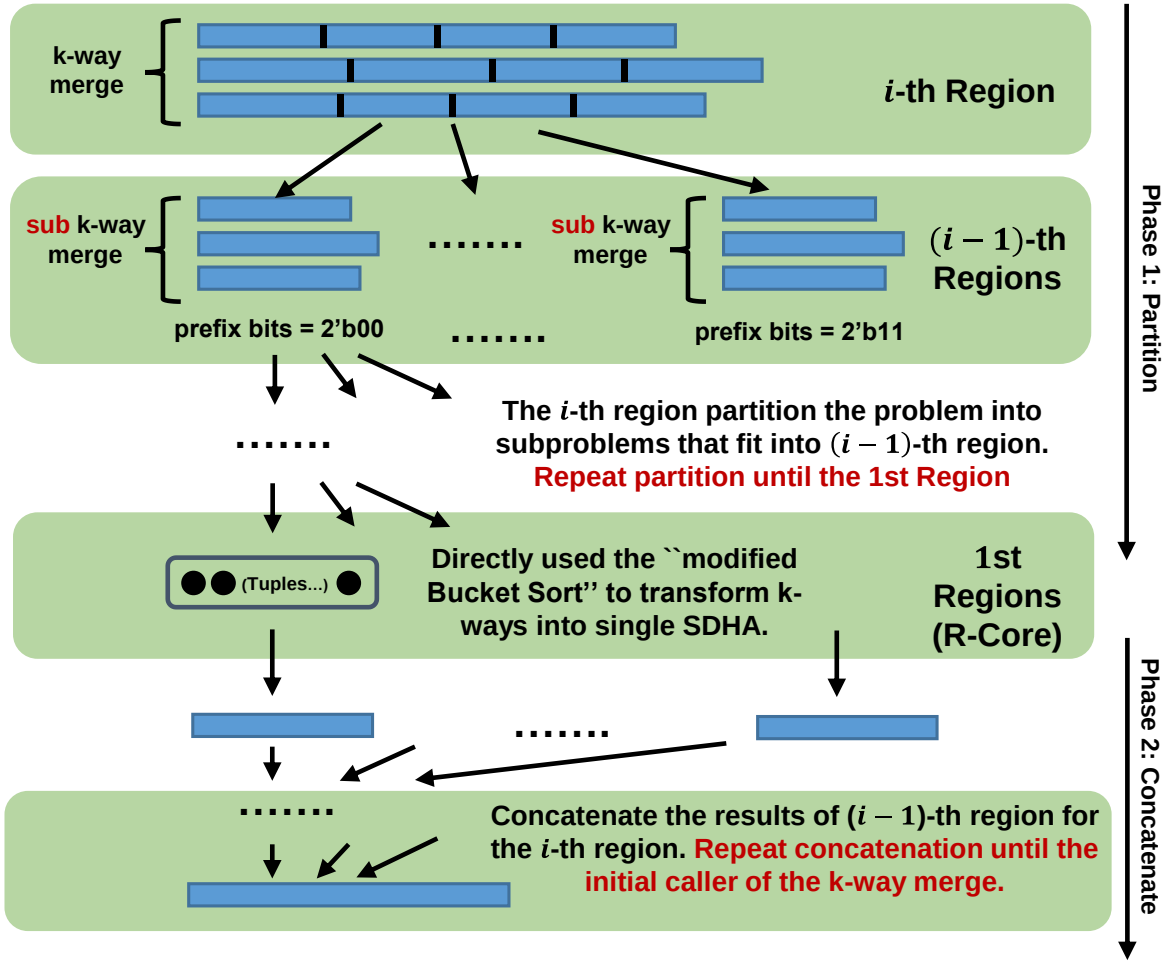


Figure 4.4: Multi-way merge: Split a multi-way merge into independent 2^q subproblems using the q prefix bits of the hash value of keys, and recursively repeat this procedure until the smallest region.

problem to make them small enough so that the 1st region (the smallest region) can solve it using the mixed Bucket Sort&Hash Table similar to the last Subsection-4.3.2, which will give us a linear time complexity. In this step, we just treat the k segments to merge as one unsorted array by simply concatenating them.

There are two problems to address for above scheme to work.

First, when we divide an SDHA array into 2^q segments using its q prefix bits of the hash value, the prefix bits become a constant in each segment and the hash values are no longer uniform enough. To make the 2^q subproblems look the same as the standard k-way merge of SDHA arrays, they should temporarily strip the first p-bits of the hash value and the rest of bits starting from $p+1$-th bits as the proxy hash value.

Second, the i-th region needs to select a q to ensure the 2^q subproblem is small enough to be solvable by the $(i-1)$-th region. This requires the total length of the k segments in every subproblem to be smaller than the size of $(i-1)$-th region's capacity (c_{i-1}). We can select $q = ceil(\log_2((\sum_i L_i)/(1/2 \times c_{i-1})))$, then the average input length for each subproblem is $1/2 \times c_{i-1}$, which is half the threshold for the maximum allowed number of the unique hash values(thus keys). The input length for each subproblem is then approximately a Gaussian distribution with mean $1/2 \times c_{i-1}$ and standard error $\sigma = \sqrt{1/2 \times c_{i-1}}$. The probability for the input length to be larger than the exceeded threshold c_{i-1} is smaller than $exp(-c_{i-1}/4)$, which is smaller than e^{-25} if only $c_{i-1} > 100$, while usually memory capacity will be much larger than 100 tuples. Finally, although the probability is extremely low, let's assume this unfortunate case indeed happens, then we further subdivide this subproblem into two smaller problems (use the $q+1$-th bit of the hash value)).

4.3.4 Parallel version

We introduce the parallel version of the algorithm by specifying the interaction between parallel computes elements with an emphasis on isolation, load balance, and more importantly, reducing duplication of tuples privately owned by different parallel units in pursuit of a smaller total memory footprint — specifically, we not only want the memory footprint to be $O(|Key|)$ instead of $O(P \times |Key|)$ where P is the parallelism, but also don't want to pay too much in synchronization overhead for that purpose.

To have better isolation between parallel compute units, their interaction is restricted to two forms 1) they can dump SDHA arrays into a shared region, and 2) when the shared region is near full (Rule 2a), they share the responsibility to merge the SDHA arrays in the shared region (also when Rule 3 is satisfied). Because dumping SDHA arrays to the shared region and merging of them are less frequent compared to the number of input tuples, the parallel compute units works in complete isolation for most of their execution time.

The memory footprint and load-balance problem is also solved. Because Rule 2a and 3 control the duplication in the shared region via multi-way merge, the total memory footprint is independent from the increase of parallelism. Finally, to evenly divide the workload of the multi-way merge to P compute units, we will use the $\log_2(P)$ prefix bits of the hash values of SDHA arrays to divide multi-way merge problem into P almost equal-sized subproblems, exploiting the uniformness of SDHA arrays. (As a result, we will in practice use prefix bits to subdivide the multi-way problem twice in nested for different purposes. The first time we use $\log_2(P)$ prefix bit into P subproblems for workload balance. The second time is used inside the multi-way merge algorithm defined in Subsection4.3.3 which will further divide them into 2^q subproblems to make them small enough.)

The benefit of reduced memory footprint from $O(P \times |Key|)$ to $O(|Key|)$ is two foldeded. In addition to capability to handle larger problems, the less noticed aspect is the indirect reduction of memory traffic by increasing the effectiveness as a traffic filter for every layer of memory hierarchy. Consider a shared buffer with capacity c, it can filter out the traffic of tuples to the next layer of memory hierarchy whose key is contained in the set of c keys that are stored in this buffer by aggregating them. Without removing the duplication that privately owned by the P parallel units, the shared buffer can store effectively only c/P distinct keys. In contrast, removing the duplication allows it to store c distinct keys and therefore filter out more traffic to the next memory layer.

Following we present the details of the parallel algorithm. We assume our algorithm runs on hardware that organizes parallel compute elements hierarchically: the basic compute elements are grouped into small clusters, then small clusters are grouped into larger clusters, which procedure can repeat multiple times until finally, the largest cluster forms the whole system. We label them as $L_1, L_2, \ldots, L_T$ clusters from small to large. For convenience, the L_1 cluster is defined to be the basic compute element although it is not a "cluster" in the parallel sense. For each level i, L_i contains a buffer that are shared by its children clusters and correspond to the i-th region in the sequential version of the algorithm. They works as follows:

- The input stream of tuples is distributed to the L_1 clusters.
- L_1 clusters build SDHA array using the mixed Bucket Sort&Hash Table method introduced in Subsection-4.3.2. It obeys Rule 1.
- All L_{i-1} children cluster dump SDHA arrays to their parent L_i cluster when Rule 1 or Rule 2b are triggered.
- When Rule 2a or Rule 3 applies a L_i cluster, it first demand all its subcluster to

wrap up on-going work, and invoke multi-way merge to all SDHA arrays in its buffer by distributing the workload to its subclusters.

- A L_i cluster may be demanded by its parent L_{i+1} cluster to wrap-up on-going work. It forwards the demand to its children clusters. After that, merge existing SDHA arrays and dump to its parent L_{i+1} cluster.

- A L_i cluster may be demanded by its parent L_{i+1} cluster to participate its multi-way merge problem. It first load the corresponding segments of each array to its buffer and finish the subproblem by further divide it and forwarding them to its subclusters. If L_i is L_1, it finish the multi-way merge using the mixed Bucket Sort&Hash Table.

4.4 Hardware

This section introduces the "reduce-by-key core (R-Core)" and discusses several implementation-related optimizations, including data compression and data mapping issues.

R-Core is just the smallest building block of the hardware. On top of R-Cores, we organize the system into a hierarchical structure by grouping R-Cores into multiple levels of nested clusters, and each cluster has a buffer shared by all its subclusters. We present an example configuration that we will use in the numerical evaluation in Table-4.2. However, our methodology does not have a mandatory requirement on those parameters, including the number of levels, subclusters per cluster, and the size of buffers; the selection presented in Table-4.2 is also not intended to be our main contribution, but as an illustration of concepts. When people absorb our design in domain-specific accelerators to serve as the backend for different applications (e.g., those mentioned in Section-4.1), they may choose those parameters in a way that can best preserve their

Table 4.2: An example configuration of the cluster hierarchy

Level	No. subcluster	Shared Buffer	Sub↔Sub Interconnect
L_1	(Reduce by key core)		
L_2	8	256KB SRAM	512bit Cubic
L_3	4	4MB SRAM	512bit All pairs
L_4	4	1GB HBM (256GB/s)	512bit All pairs
L_5	1	128GB DDR4 (32GB/s)	N/A

original top-level organization of those accelerators.

4.4.1 The "reduce-by-key core (R-Core)"

The "reduce-by-key core" mainly implements the modified Bucket Sort introduced in Subsection-4.3.2, which is the working horse for both SDHA building and multi-way merging. We emphasize squeezing the number of cycles for the basic operations in the design, including inserting and dumping tuples.

Bucket Table: The central part is the bucket table and its access logic. To reduce the average probe time in insertion, we need to reduce the chance multiple tuples fall into the same bucket, which requires making the table more empty by either increasing the bucket table size or reducing the number of tuples stored in the table. To avoid wasting precious SRAM, we store the tuples and the meta-data of the bucket table (mainly pointers of the link list) separably (The "Pointers" and "Tuples"in yellow box in Figure-4.5-a). The pointers' bit-width is small since a tiny SRAM's address space is very small, so we decide it is affordable to make the bucket table 4 times larger than the maximum number of tuples to store. We consider the following configuration for the rest discussion:

1. We store maximum $c = 1k$ tuples, each tuple is $W = 8$Byte long.

2. The bucket table has $4c_i$ entries and is stored in an SRAM with $(5 + 1)c$ entries, each entry is $\log_2(c) = 10$bits. The first $4c_i$ entries store the pointer toward the

first tuple of the $4c_i$ bucket. For $0 \leq x < c$, the $4c_i + x$ entry is the pointer to the next tuple of the tuple stored in position i.

To efficiently iterate through all non-empty buckets when we dump all tuples, we also store a two-level bitmap to mark all non-empty buckets (The "Bitmap" in the yellow box in Figure-4.5-a). The first level is a $\sqrt{4c}$ bitmap whose i-th bit indicates whether any of the $i\sqrt{4c} \sim i\sqrt{4c}-1$ buckets non-empty. The second level is an array of $\sqrt{4c}$ bitmap, each

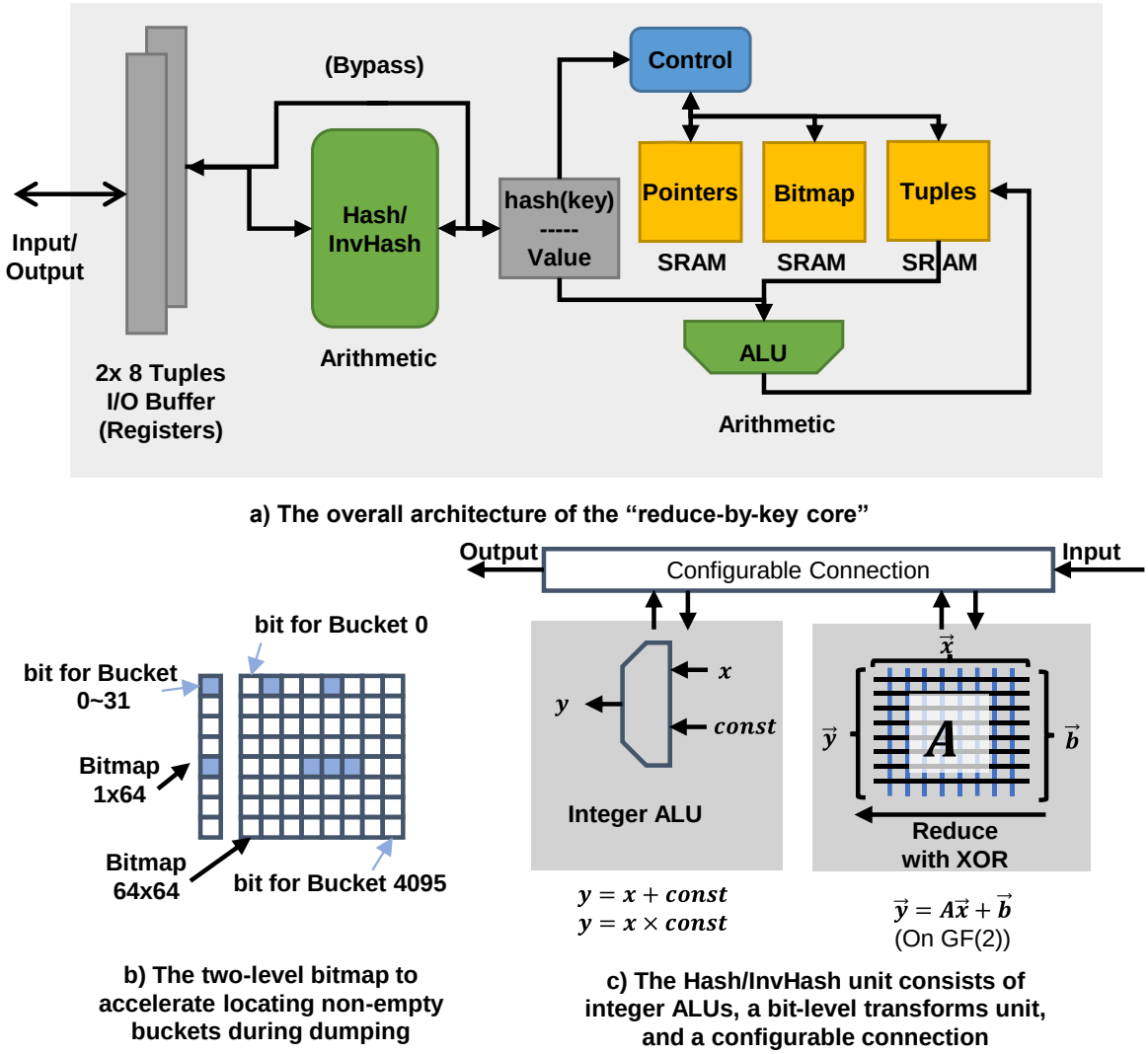


Figure 4.5: The detailed architecture of RCore with Bucket Table and Hash/Inverse Hash.

containing $\sqrt{4c}$ bits indicating the emptiness of every bucket (Figure-4.5-b). To achieve a near one-tuple-per-cycle insertion throughput in the pipeline, several operations must be executed simultaneously: probing the bucket table to obtain the pointer, reading the hash value field, reading the value field, updating the value field of the last tuple with the result from the ALU, and updating the linked-list in the bucket by moving the last tuple to the head of its bucket. Consequently, we partition the SRAM into several segments: an 8KB 1R1W SRAM for tuples, a 5KB 1R1W SRAM for the "head pointer" of the bucket table, a 1.25KB 2RW SRAM for the "next pointer" of the bucket table, and a 0.5KB bitmap. In summary, each "reduce-by-key core" requires a total of 14.75KB of SRAM. All operations are pipelined, yet particular attention must be given to read-after-write hazards. These hazards can arise when reordering the linked list in the bucket table if an earlier pipeline stage contains a tuple in the same bucket and is traversing the linked list. We note that when such a hazard occurs, there are two possible scenarios:

1. The two tuples in the same bucket share an identical key; thus, the earlier tuple can forward its address and value to the subsequent tuple.

2. The two tuples in the same bucket possess distinct keys; consequently, the later tuple must await the completion of the update to the linked list by the earlier tuple.

The forwarding mechanism resolves the hazard in the first scenario, and the second scenario is fortunately infrequent (as elucidated below), allowing the pipeline to achieve nearly one-tuple-per-cycle throughput.

The rarity of the second scenario arises due to the following reasons:

1. A read-after-write hazard does not occur if the keys of the two tuples are mapped to different buckets.

2. A read-after-write hazard is not encountered if the two tuples are spaced far apart in the input stream, meaning their distance surpasses the pipeline depth. This situation can only arise when a majority of the input tuples cluster around a few keys, and these keys coincidentally map to the same bucket.

3. The hash function ensures a uniform distribution of different keys across distinct buckets.

Given the above three factors, it can be inferred that a read-after-write hazard will only materialize when a specific key overwhelmingly represents the majority of input keys. In such cases, forwarding can effectively mitigate the hazard.

Hash and Inverse Hash: The "reduce-by-key core" need to implement an invertible hash function in both directions, as we need to compute the hash value of a key in the SDHA building phase and recover the key from the hash value in the epilogue. A hash function is invertible if it is a composition of the following primitives, and its inverse is the composition of the inverse of each primitive in reversed order:

- hash = hash * odd constant; (with truncate)
- hash = hash + constant; (with truncate)
- $\overrightarrow{hash} = A \times \overrightarrow{hash} + \overrightarrow{b}$; (Here, $\overrightarrow{hash}$ is the bit vector representation of hash. A is an invertible matrix with elements in finite field $GF(2)$)

The third term above is the generalization of the following operations:

```
hash ^= constant;
hash = some permutation of bits(hash);
hash ^= hash >> constant;
hash ^= hash << constant;
```

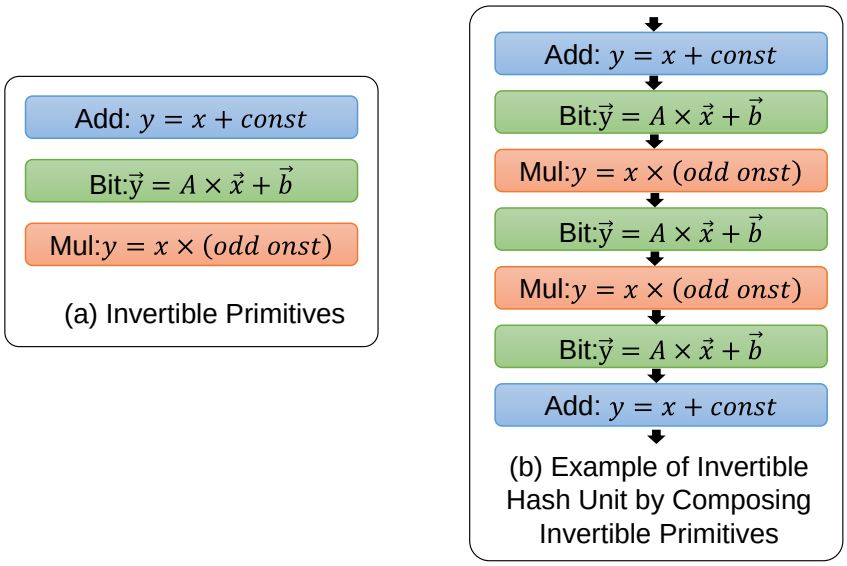


Figure 4.6: The structure of the Hash unit

The hardware implementation of the Hash/Inverse Hash unit is shown in Figure-4.5-c. It can be configurated by how the three primitives (Figure-4.6-(a)) are composed layer by layer (Figure-4.6-(b)).

4.4.2 Data compression

We discuss the opportunity and the method to compress SDHA arrays. Data compression can be used both in off-chip memory to reduce DRAM/network traffic and footprint and in on-chip to increase effective on-chip memory capacity.

The SDHA arrays are more compression-friendly than other methods like hash tables. First, the streaming access pattern makes the added latency for (de)compression nearly transparent. Second, it permits compression methods that only apply to data in chunks instead of a single element. Third, the hash field of adjacent tuples in an SDHA array has similar prefix bits and, therefore, is naturally compressible.

Consider an SDHA array with length L; due to the uniformness and monotonicity of SDHA arrays, the first $\log_2(L)$bit of the hash field of two adjacent tuples is likely to be

the same or differ by just one. This allows us to save up to $\log_2(L)$bit per tuple if the proper compression method is used. We have two compression methods as follows; let's define $r = ceil(\log_2(L))$ and $L^* = 2^r$, so $L^* \leq L < 2L^*$.

- Method 1 (faster (de-)compression): Use the $(r-5)$ prefix bits of the hash value to divide the SDHA into chunks of $32 \sim 64$ tuples. We create an auxiliary array to store the offset to each compressed chunk. For each such chunk, we don't store the common $(r-5)$ prefix bits in tuples to save space, which can be infered by the chunk id implicitly. The offset for each chunk cause ~ 1 bit amortized overhead per tuple, so we save $(r-6)$ bits per tuple[1]. The (de)-compression is highly parallel and simple.

- Method 2 (higher compression ratio): We strip the r prefix bits of the hash values to save space. We use two bitmaps to recover them. The first bitmap has length L, and the i-th bits indicate whether the i-th tuple's r prefix bits of the hash value are identical to the prior tuple. The second bitmap has length L^*, and the i-th bits indicate whether there is at least one tuple whose r prefix bits equals i. Then, we also use the $(r-10)$ prefix bits to divide the SDHA array into chunks of $1024 \sim 2048$ tuples and use an auxiliary array to store the offset. In this way, we save about $(r-2)$ bits per tuple. The (de-)compression of each chunk is more complicated.

This saving becomes significant when the SDHA array is long. For example, an off-chip SDHA array usually has $L \sim 1M$, then $\sim 18bit$ can be saved per tuple, which is $\sim 28\%$ reduction for an 8byte tuple. In addition, this saving is unconditional to applications.

The value field of the tuple might also be compressible, but this opportunity is application dependent. For counting-like applications, such as the k-mer counting in genome

[1]The number 5 is the best choice under this scheme

analysis, most values are small integers like 1 or 2. They are also suitable for variable-length integer encoding.

In hardware, use Method 1 to compress SDHA arrays on-chip and Method 2 to compress SDHA arrays in off-chip memory.

4.5 Analysis

We group the buffers according to their capacity and list their capacity from small to big as $c_1, c_2, \ldots, c_L$ and the number of buffers with capacity c_i as n_i. c_i also correspond the capacity of i-th region.

Theorem 4.5.1. The peak memory usage MaxUse_i in the buffer with capacity c_i is bounded by the following equation:

$$\text{MaxUse}_i \leq \min(c_i, (1+c) \cdot |Key| + c_{i-1})$$

Proof. Since an SDHA array can not have duplicated keys, any SDHA array can't be longer than the number of unique keys known from the input.

If the buffer had never spilled any SDHA array before, then either it is empty, or it still contains the longest SDHA array that had ever appeared in this buffer. In the first case, the memory usage is 0. In the second case, suppose the length of that array is l, then a Rule 3 merge will be triggered when the memory usage exceeds $(1+c)l$. The peak memory usage is reached when its usage is slightly below $(1+c)l$ while one of child spilled an SDHA array to it, whose length l' is bounded by c_{i-1}. Then the memory usage will be $(1+c)l + l' \leq |Key| + c_{i-1}$.

If the buffer had spilled some SDHA arrays before, then $|Key| \geq 2/3c_i$ and Rule 3 will have no impact on this buffer since the threshold to trigger Rule 3 is already lower

than Rule 2a, as $(1+c) \cdot 2/3c_i \geq c_i$ for $c \geq 1$. The peak memory usage is bounded by c_i.

Therefore, the peak memory usage is bounded by the minimum of c_i and $(1+c) \cdot |Key| + c_{i-1}$. □

Corollary 4.5.2. The total memory usage

$$\begin{aligned}
\text{TotalMemUse} &\coloneqq \sum_{i=1}^{L} n_i \cdot \text{MaxUse}_i \\
&\leq \sum_{i=1}^{L} n_i \cdot \min(c_i, (1+c) \cdot |Key| + c_{i-1}) \\
&\leq C_< + p_\geq \cdot (\frac{5}{2} + c) \cdot |Key| \\
&= O(|Key|)
\end{aligned}$$

where $C_<$ is the total capacity of all memory regions whose capacity is smaller than $\frac{3}{2}|Key|$, and $p_\geq$ is the number of smallest level of clusters whose buffer is just larger than $\frac{3}{2}|Key|$.

In real hardware, the lower-layer memory is usually significantly larger than the upper-layer memory. For example, the L1 cache is usually 32KB, L2 cache is 256KB, L3 cache is 12MB on a typical desktop CPU as of 2022. Therefore we can approximately consider the total memory footprint as the sum of those largest nonempty regions, which is $p_\geq \cdot (\frac{5}{2} + c) \cdot |Key|$.

Notice the dependency of the total memory footprint on parallelism. The multiplicative factor $p_\geq$ can be greater than 1 when $|Key|$ is small, but it reduces to 1 when $|Key|$ is large. We usually care about space efficiency only when the workload consumes a lot of memory; therefore, $p_\geq$ is harmless as it is 1 when it is matters.

Finally, since as $|Key|$ grows $p_\geq \to 1$ (only the largest buffer), the asymptotic memory footprint is $O(|Key|)$.

4.5.1 Access grainularity

All data traffic between memory hierarchy levels is either for dumping an SDHA array or merging multiple SDHA arrays. Both are reading/writing long continuous array access instead of randomly scattered tuples, and their lengths are more than sufficient to obtain the full bandwidth from the memory system.

First, we consider dumping an SDHA array from the i-th region to the next$(i+1)$-th region. The length of the SHDA array is at least 2/3 of the maximum capacity of the i-th region (according to rules 1, 2a, 2b).

Second, we analyze the granularity of memory access during a multi-way merge. The access granularity is the segments of the k-array. By taking $q = ceil(\log_2((\sum_i L_i)/(1/2 \times c_{i-1})))$ required in Subsection-4.3.3, the average length of a segment is $c_{i-1}/(2k)$. The k can't be arbitrarily large because the SDHA arrays at the i-th regions are either dumped from the $(i-1)$-th region or the result of a multi-way merge of them; and according to Rule 2b, their length is at least $2/3 \times c_{i-1}$. Then we have $k \leq c_i/(2/3 \times c_{i-1})$. We finally derive that the memory access granularity is at least $c_{i-1}^2/(3c_i)$.

For illustration, let's suppose $c_{i-1} = 1M$ tuples (SRAM), $c_i = 1G$ tuples (DRAM). Then the access granularity to the DRAM is 1M tuples to dump an SHDA array (write operation) and 333 tuples to merge SDHA arrays (read operation). If each tuple is 8bytes, then memory access will be at least 2.6KB of continuous data, which is enough to be considered as streaming access for an 8GB ($c_i \times 8Byte$) DRAM.

4.6 Memory Traffic

We can prove that the data traffic caused by our algorithm will not be significantly worse than any possible algorithm. However, there are still some ambiguities in this description. For example, we need to carefully define what we mean by "any possible

algorithm."

To address this, in the rest of this section, we will proceed in the following order:

1. We provide definitions for "hardware" and "legal execution paths.". We define an algorithm as a function that maps inputs to execution paths, enabling us to discuss the set of "all possible algorithms."

2. We examine a simple two-level memory hierarchy hardware. On this hardware, the memory traffic caused by our algorithm will not be significantly worse than any possible algorithm, and in particular, the subset of all practical algorithms that do not depend on the future input information.

3. We generalize our results to hardware with multiple levels of the memory hierarchy.

4.6.1 Define Hardware and Legal Execution Paths

We take a inpue sequence I that consists of $|I|$ tuples $[(k_1, v_1), (k_2, v_2), \ldots, (k_{|I|}, v_{|I|})]$. For all i, $k_i \in Keys$ and $v_i \in Values$. For convenience, we can assume all elements in $Keys$ have appeared in I at least once[2].

We consider hardware with an on-chip memory called "Cache" and an off-chip memory "DRAM". The on-chip memory has a finite capacity C, and the off-chip memory has an infinite capacity (but countable). Aggregation operation can only be applied to two tuples on Cache, and the result must also be written to Cache. Each memory slot can hold up to one tuple or be "Empty". We label each memory slot with an integer,

[2]otherwise, we can remove it from the $Keys$

specifically as follows:

$$\text{Cache} = \{1, 2, \ldots, C\}$$

$$\text{DRAM} = \{C + 1, C + 2, \ldots\}$$

$$\text{so Cache} \cup \text{DRAM} = \mathbb{N}^+$$

The memory state can be encoded as a map from positive integers to the stored content, which is either a tuple or Empty.

$$MemState : \mathbb{N}^+ \to (Keys \times Values) \cup \{\text{Empty}\}$$

At any moment, the state of the remaining input sequence is a sequence of tuples. We define the $InputState$ as the set of all possible remaining input sequences, which is $(Keys \times Values)^+$

$$InputState = (Keys \times Values)^+$$

When hardware is executing, its overall state consists of the memory state and the remaining input sequence:

$$SystemState : MemState \times InputState$$

The hardware can perform the reduce-by-key operation because it executes a sequence

of actions that move the hardware from one state to another.

$$state_0 \rightarrow state_1 \rightarrow state_2 \rightarrow \cdots \rightarrow state_n$$

Definition 4.6.1. The final state $state_n$ is said to be success to finish the input sequence when it satisfies the following conditions:

1. The remaining input sequence is empty.
2. The memory slots in Cache are all empty.
3. Each key k in $Keys$ appears exactly once in the memory slots in DRAM. The value of the tuple in the memory slot is the sum of all values of tuples in I that have key k.

$$\begin{aligned} Sucess_I(state) = \text{let: } & (\text{mem, rest}) = \text{state in} \\ & rest = \emptyset \\ & \wedge \forall i \in \text{Cache}, mem(i) = \text{Empty} \\ & \wedge \forall k \in Keys, \text{exist and only exist one } i \in \text{DRAM}, mem(i) = (k, v_k) \\ & \text{where } v_k \in Values, \text{and } v_k \text{ is the sum of all } v \text{ in tuples from } I \text{who has key } k \end{aligned}$$

Now we look at actions that trigger state transitions: $state_i \rightarrow state_{i+1}$. All possible actions are lised as follows:

Definition 4.6.2. Possible actions to perform on a hardware at state $state_i = (mem, rest)$:

1. Load(s, d), load a tuple from DRAM position s to Cache position d.
   (a) Prequeisites 1: $s \in$ DRAM and $d \in$ Cache.

(b) Prerequisites 2: $mem(s) \neq$ Empty, and $mem(d) =$ Empty.

2. Store(s, d), store a tuple from Cache position s to DRAM position d.

   (a) Prequeisites 1: $s \in$ Cache and $d \in$ DRAM.

   (b) Prerequisites 2: $mem(s) \neq$ Empty, and $mem(d) =$ Empty.

3. Input(s), take a tuple from the remaining input sequence and store it in Cache position s.

   (a) Prequeisites 1: $s \in$ Cache.

   (b) Prerequisites 2: $mem(s) =$ Empty.

   (c) Prerequisites 3: $rest \neq \emptyset$.

4. Aggegate(s_1, s_2, d), aggregate two tuples in Cache positions s_1 and s_2 and store the result in Cache position d.

   (a) Prequeisites 1: $s_1, s_2, d \in$ Cache.

   (b) Prequeisites 2: $mem(s_1) \neq$ Empty, $mem(s_2) \neq$ Empty, and $mem(d) =$ Empty.

   (c) Prequeisites 3: $mem(s_1).key = mem(s_2).key$.

Definition 4.6.3. A sequence of actions $A = [a_1, a_2, \ldots, a_n]$ is said to be legal for a state sequence $S = [state_0, state_1, \ldots, state_n]$ if it satisfies the following conditions:

1. $state_0$ is the initial state, the memory slots in Cache are all empty, and the remaining input sequence is I. For each key, there is exactly one tuple in the memory slots in DRAM, storing the initial values for each key, like zero. In particular, if, in our algorithm, we require those initial tuples to be organized in the layout as an SDHA array.

2. $state_n$ is successful in finishing the input sequence.

3. For all i, a_i is applicable to $state_i$.

4. For all i, $state_{i+1} = execute(state_i, a_i)$.

The execute function can be defined using the description of actions in the previous definition, and we do not repeat it here.

There is a design choice about the initial state. Alternative to above definition, we can also require the memory slots in DRAM to be empty. This difference makes little difference if the input sequence is long enough. But the current definition makes the initial state and the final state symmetry, and the benefit is the number of load(s,t) and store(s,t) actions are precisely the same.

Lemma 4.6.1. If a legal sequence of actions A is executed on hardware, then the traffic of the hardware is the same as the number of $Load(s, d)$ and $Store(s, d)$ actions in A.

$$|\{a \in A | a \text{ is } Load\}| = |\{a \in A | a \text{ is } Store\}|$$

Proof. We count the DRAM usage as the number of memory slots in DRAM that are not empty.

$$\text{DRAM usage} = |\{\imath \in \text{DRAM} | mem[i] \neq Empty\}|$$

We only need to notice that $Load(s, d)$ decreases the DRAM usage by 1, and $Store(s, d)$ increases the DRAM usage by 1. The other actions do not change the DRAM usage. Both the initial state and the final state have the same DRAM usage, which is $|Key|$. Therefore the number of $Load(s, d)$ and $Store(s, d)$ actions in A is the same. □

Now we can define the traffic of the hardware when it executes a legal sequence of actions. It is the number of $Load(s, d)$ and $Store(s, d)$ actions that are performed on the hardware.

Definition 4.6.4. The traffic of a legal sequence of actions A is defined as:

$$\text{Traffic}(A) \coloneqq \sum_{a \in A} \text{if } \ a \text{ is Load or Store, then } \ 1 \text{ else } 0$$

The store traffic is the number of $Store(s, d)$ actions that are performed on the hardware.

$$\text{Store\_traffic}(A) \coloneqq \sum_{a \in A} \text{if } \ a \text{ is Store, then } \ 1 \text{ else } 0$$

With Lemma-4.6.1, we have:

$$\text{Traffic}(A) = 2 \times \text{Store\_traffic}(A)$$

Definition 4.6.5. We can define the legal set of actions $\text{Legal}(C, I)$ according to the input sequence I and the number of memory slots in Cache, C.

$\text{Legal}(C, I) \coloneqq \{\text{legal sequences of actions that finished } I \text{ and use less than } C \text{ cache size}\}$

Definition 4.6.6. The optimal traffic is defined as the minimum possible traffic of a legal sequence of actions.

$$\text{Opt}(C, I) \coloneqq \min_{A \in \text{Legal}(C,I)} \text{Traffic}(A)$$

We can now properly define algorithm.

Definition 4.6.7. An algorithm Alg is a function that maps a cache size C and an input

sequence I to a legal sequence of actions A in Legal(C, I).

$$\text{Alg} : \mathbb{N} \times (Keys \times Values)^+ \rightarrow \text{Actions}^+$$
$$\text{with Alg}(C, I) \in \text{Legal}(C, I)$$

We usually care more about a subset of algorithms that are practical in engineering. That is, the algorithm is deterministic, which means it should not depend on future information. We can define the deterministic algorithm as follows.

Definition 4.6.8. An algorithm Alg is said to be deterministic if its first j input tuples fully determine the actions before the $j+1$-th Input-action.

Specifically, for any to input sequence I_1 and I_2 that are the same before the j-th tuple, their actions, $A_1 = \text{Alg}(C, I_1)$ and $A_2 = \text{Alg}(C, I_2)$, are the same before the $j+1$-th action.

$$\forall I_1, I_2 \in (Keys \times Values)^+ \quad \forall j \in \mathbb{N} \text{ with } j < |I_1|, j < |I_2|$$
$$I_1[1:j] = I_2[1:j] \implies \text{the prefix of } A_1 = \text{Alg}(C, I_1) \text{ and } A_2 = \text{Alg}(C, I_2)$$
$$\text{are the same upto the } j+1\text{-th Input action}$$

4.6.2 Two-level system

We first briefly review the data traffic generated by different steps in our algorithm. Then we focus on the dump-related traffic and compute an upper bound for this type of traffic. It is then used to set the upper bound for the total traffic of the algorithm.

1. When a new input tuple x_t arrives if the on-chip memory is not full, then it is inserted into the on-chip memory.

2. After the intersection, if the on-chip memory is full, then all content in the buffer

is fully deduplicated. (Rule 2a)

3. After deduplication, if the on-chip memory is still at least 2/3 full, then all the content in the buffer is dumped to the off-chip memory in the form of a SDHA array. This step may cause traffic (many Store(s,d)) from the on-chip memory to the off-chip memory. (Rule 2b)

4. If the total usage of the off-chip memory exceeds the longest SDHA array that had ever appeared by a factor of $(1 + c)$ (here, c is a small constant, such as some value between 1 and 2), then all SDHA arrays in the off-chip memory are merged into a single SDHA array. Since aggregation can only happen on-chip, those arrays need to be streamingly loaded on-chip, and the result needs to be stored back. This step may cause traffic (Load(s,d)) from the off-chip memory to the on-chip memory and traffic (Store(s,d)) from the on-chip memory to the off-chip memory. (Rule 3).

5. Otherwise, continue to process the next input tuple.

6. Finally, after there are no rest input tuples, we start the post process that aggregates all remaining tuples in both on-chip and off-chip memory: it performs full-deduplication on-chip, then dump it to off-chip memory, and then performs a multi-way merge. This step causes both Load(s,d) and Store(s,d) traffic.

We refer to the traffic in step 3 as dump-related traffic (denoted as Spill) and the traffic in step 4 as merge-related upward/downward traffic (denoted as Merge_Up and Merge_Down respectively). The traffic in step 6 can be considered an extra, final, and unconditionally part of dump-related + merge-related traffic.

Dump-related traffic

Theorem 4.6.2. The total dump-related traffic Spill, when running our algorithm on a machine with on-chip memory capacity C for input I, is bounded by the optimal traffic in the following way:

$$\text{Spill} \leq \min_{0<r<1} \frac{1}{2} \cdot \frac{1}{1-r} \cdot \text{Opt}(\frac{2r}{3} \cdot C, I)$$

Proof. We let our algorithm Alg_{our} run on a machine with capacity C for input I. The action sequence it executes is:

$$A_{our} = \text{Alg}_{our}(C, I) \in \text{Legal}(C, I)$$

At the same time, for all r that have $0 < r < 1$, we consider a hypothetical machine with a slightly smaller capacity $\frac{2r}{3} \cdot C$. This machine runs the optimal action sequence $A_{opt} \in \text{Legal}(\frac{2r}{3} \cdot C, I)$:

$$A_{opt} = \underset{A \in \text{Legal}(\frac{2r}{3} \cdot C, I)}{\arg\min} \text{Traffic}(A)$$

A_{our} and A_{opt} have an exactly equal number of $Input$ actions, which also match the number of input tuples $|I|$.

Now, let's denote the number of times that dumps are triggered by Rule 2b as K. The k-th dump is triggered after t_k-th input tuple. So they are triggered at the $t_1, t_2, \ldots, t_K$-th input tuple. We set $t_0 = 0$ and $t_{K+1} = |I|$. (Then we have $t_0 < t_1 < t_2 < \ldots < t_K < t_{K+1}$).

We divide A_{our} into $K + 1$ segments, where the j-th segment ($1 \leq j \leq k + 1$) covers the actions after (inclusive) the t_j-th Input action and before (exclusive) the t_{j+1}-th Input

action. We denote those segments as A^j_{our}.

$$A_{our} = A^1_{our} + +A^2_{our} + + \ldots + +A^{K+1}_{our}$$

like following

$$\underbrace{\underbrace{Input(\_)}_{\text{1st}}, \cdots, \overbrace{\underbrace{Input(\_)}_{t_j\text{-th}}, \cdots, \underbrace{Input(\_)}_{t_j+1\text{-th}}}^{A^j_{our}}, \underbrace{Input(\_)}_{t_{j+1}\text{-th}}, \cdots, \underbrace{Input(\_)}_{|I|\text{-th}}, \cdots}_{A_{our}}$$

Similarly, we divide A_{opt} into $K+1$ segments, using the same $t_1, t_2, \ldots, t_K$ as the boundaries. We denote those segments as A^j_{opt}.

$$A_{opt} = A^1_{opt} + +A^2_{opt} + + \ldots + +A^{K+1}_{opt}$$

like following

$$\underbrace{\underbrace{Input(\_)}_{\text{1st}}, \cdots, \overbrace{\underbrace{Input(\_)}_{t_j\text{-th}}, \cdots, \underbrace{Input(\_)}_{t_j+1\text{-th}}}^{A^j_{opt}}, \underbrace{Input(\_)}_{t_{j+1}\text{-th}}, \cdots, \underbrace{Input(\_)}_{|I|\text{-th}}, \cdots}_{A_{opt}}$$

We denote the dump-related traffic of our algorithm's j-th segment (A^j_{our}) as Spill_j, and the store traffic of the optimal action sequence's the j-th segment (A^j_{opt}) as Opt^j_{Store}, the total store traffic of the optimal action sequence as Opt_{Store}. Then we have:

$$\text{Spill} = \sum_{j=1}^{k+1} \text{Spill}_j$$
$$\text{Opt}_{Store} = \sum_{j=1}^{k+1} \text{Opt}_{Store}$$

In addition, applying Lemma-4.6.1, we know that for any legal action sequence, the total

traffic is twice the store traffic. Therefore, we have:

$$\text{Opt}(\frac{2r}{3} \cdot C, I) = 2 \times \text{Opt}_{Store}$$

In the $K+1$ segments, the first k segments are triggered by Rule 2b, and the last segment is part of the post-process and is unconditional. We discuss them separately.

For the first k segment, according to Rule 2b, those dumps are triggered only when there is still more than $\frac{2r}{3} \cdot C$ tuples in the on-chip memory after performing full deduplication, and those tuples will be dumped to the off-chip memory as a whole. Therefore, we have:

$$\text{Spill}_j \geq \frac{2r}{3} \cdot C \text{if } j < K+1$$

Meanwhile, in our algorithm, each segment starts with an empty on-chip memory; this is because the last segment dumped all the tuples in the on-chip memory into the off-chip memory. Therefore, in order to make this dump happen, this segment must take at least $\frac{2r}{3} \cdot C$ tuples as input to have so many distinct keys on-chip. Therefore, the slice of input sequence I from t_j to t_{j+1} must have at least $\frac{2r}{3} \cdot C$ distinct keys in its key field. i.e.

$$|\{I[t_j].keys, I[t_j+1].keys, \ldots, I[t_{j+1}-1].keys\}| \geq \text{Spill}_j \geq \frac{2r}{3} \cdot C$$

Now consider the j-th segment of A_{opt} in the optimal action sequence, i.e. A^j_{opt}. It reads the same set of input tuples as our algorithm's j-th segment A^j_{our}. Therefore, it also contains the Input actions that bring the input from $I[t_j]$ to $I[t_{j+1}-1]$ to the on-chip memory. There must be at least $\text{Spill}_j \geq \frac{2r}{3} \cdot C$ distinct keys in their key field. But since it's on-chip memory capacity is only $\frac{2r}{3} \cdot C$, and the only action that can reduce the number

of distinct keys in the on-chip memory is Store(s,d), so the optimal action sequence must contain at least $\left(\text{Spill} - \frac{2r}{3} \cdot C\right)$ Store(s,d) actions in A^j_{opt}. (Note: Aggregate(s_1, s_2, d)) only combine tuples with the same key).

Therefore, we have:

$$\text{Opt}^j_{Store} \geq \text{Spill} - \frac{2r}{3} \cdot C$$

And we have $\forall j \in \{1, 2, \ldots, k\}$:

$$\begin{aligned}
\frac{\text{Opt}^j_{Store}}{\text{Spill}_j} &\geq \frac{\text{Spill} - \frac{2r}{3} \cdot C}{\text{Spill}_j} \\
&\geq \frac{\frac{2}{3} \cdot C - \frac{2r}{3} \cdot C}{\frac{2}{3} \cdot C} \\
&= 1 - r
\end{aligned}$$

For the last segment, the number of unique keys in the corresponding part of input sequence I is exactly Spill_{k+1}. At the same time, remember that after the last segment of optimal action sequence A^{k+1}_{opt}, the on-chip memory must be empty, therefore A^{k+1}_{opt} must contain at least Spill_{k+1} Store(s,d) actions. Therefore, we have:

$$\begin{aligned}
\frac{\text{Opt}^{k+1}_{Store}}{\text{Spill}_{k+1}} &\geq \frac{\text{Spill}_{k+1}}{\text{Spill}_{k+1}} \\
&= 1
\end{aligned}$$

Then, with $0 < r < 1$,

$$\begin{aligned}\frac{\mathrm{Opt}(\frac{2r}{3} \cdot C, I)}{\mathrm{Spill}} &= \frac{2 \cdot \mathrm{Opt}_{Store}}{\mathrm{Spill}} \\ &= \frac{2 \cdot \sum_{j=1}^{K+1} \mathrm{Opt}_{Store}^{j}}{\sum_{j=1}^{K+1} \mathrm{Spill}_j} \\ &\geq \frac{2 \cdot \sum_{j=1}^{K+1} \mathrm{Spill}_j \cdot (1-r)}{\sum_{j=1}^{K+1} \mathrm{Spill}_j} \\ &= 2 \cdot (1-r)\end{aligned}$$

Now, take all possible r in range $(0, 1)$ into consideration, we have:

$$\mathrm{Spill} \leq \min_{0<r<1} \frac{1}{2} \cdot \frac{1}{1-r} \cdot \mathrm{Opt}(\frac{2r}{3} \cdot C, I)$$

□

Merge-related Traffic

Now we shift our focus to the merge-related traffic.

We denote the merge-related upward/downward traffic as Merge_Up and Merge_Down, respectively. Then we have:

Theorem 4.6.3. The merge-related downward traffic is bounded by:

$$\mathrm{Merge\_Down} \leq \frac{1}{c} \cdot \mathrm{Spill} + |Key|$$

where c is the const threshold parameter used in Rule 3.

Proof. We only need to observe that since we have an initial SDHA array in off-chip memory with length $|Key|$ and, after every multi-way merge, the length of the result

SDHA array is still not changed since it will contain at least $|Key|$ distinct keys and also at most $|Key|$ distinct keys.

To trigger a multi-way merge (except the last one), the usage of off-chip usage must exceed $(1 + c) \cdot |Key|$. After each multi-way merge, the result off-chip usage is $|Key$, which means at least $c \cdot |Key|$ tuples are spilled to the off-chip memory since the last merge. Therefore, the number of times that multi-way merges are triggered $n_{\text{Multi-Merge}}$ (including the last unconditional one in past process) is bounded:

$$n_{\text{Multi-Merge}} \leq \frac{\text{Spill}}{c \cdot |Key|} + 1$$

Since in each merge, the result SDHA array has length $|Key|$, the merge-related downward traffic is bounded by:

$$\text{Merge\_Down} \leq n_{\text{Multi-Merge}} \cdot |Key| \leq \frac{1}{c} \cdot \text{Spill} + |Key|$$

□

Then we can compute the total traffic of our algorithm:

Theorem 4.6.4. The total traffic of our algorithm is bounded by:

$$\text{Total}_{\text{our}} \leq \min_{0<r<1} \frac{1+c}{c} \cdot \frac{1}{1-r} \cdot \text{Opt}(\frac{2r}{3} \cdot C, I) + 2 \cdot |Key|$$

Proof. The total traffic of our algorithm is the sum of the spill-related traffic and the merge-related traffic:

$$\text{Total}_{\text{our}} = \underbrace{\text{Spill} + \text{Merge\_Down}}_{\text{As Store actions}} + \underbrace{\text{Merge\_Up}}_{\text{As Load actions}}$$

Due to Lemma-4.6.1, the number of Load actions are equal to that of Store actions, therefore:

$$\begin{aligned}
\text{Total}_{\text{our}} &= \text{Spill} + \text{Merge_Down} + \text{Merge_Up} \\
&= 2 \cdot (\text{Spill} + \text{Merge_Down}) \\
&\leq 2 \cdot \left(\text{Spill} + \frac{1}{c} \cdot \text{Spill} + |Key| \right) \\
&\leq \min_{0<r<1} \frac{1+c}{c} \cdot \frac{1}{1-r} \cdot \text{Opt}(\frac{2r}{3} \cdot C, I) + 2 \cdot |Key|
\end{aligned}$$

□

Discussion

Consider the upper-bound in Theorem-4.6.4, the direction of $\min_{0<r<1}$ term means the effective upper-bound is the best (smallest) one for all possible r.

It's not obvious to see which r is the effective r without knowing the exact input sequence I. However, we can still get some insights.

1. When $r \to 1$, the term $\frac{1}{1-r}$ will increase dramatically; on the otherside, the term $\text{Opt}(\frac{2r}{3} \cdot C, I)$ is decreasing. In particular, if the number of actively used keys is smaller than the threshold $\frac{2r}{3} \cdot C$, the term $\text{Opt}(\frac{2r}{3} \cdot C, I)$ will be very close to zero. Therefore, for input sequence with small working set, the effective upper-bound took when r is close to 1.

2. When $r \to 0$, the term $\frac{1}{1-r}$ will be minimized (being 1); on the other side, the term $\text{Opt}(\frac{2r}{3} \cdot C, I)$ is increasing as it looks like a machine with very small cache size, and all traffic are spilled (in a sense like all cache access is missing) and $\text{Opt}(\frac{2r}{3} \cdot C, I)$

is approximately the input size. Therefore, for input sequence with large working set such that the small on-chip memory size is ineffective anyway, the effective upper-bound took when r is close to 0.

3. Finally, r is not a design parameter. We don't set it's value. It only appear in analysis at automatically choose the best one to be the effective upperbound.

The second term in upper-bound, $2 \cdot |Key|$, is contributed by the final post-process. It is a constant that is independent with the length of input sequence. It can become negligible when the length of input sequence scales to a sufficient length (e.g. repeat the input sequence for 100 times). In particular, it is trivial to see that any algorithm need to produce at least $2 \cdot |Key|$:

$$\forall X, Opt(X, I) \geq 2 \cdot |Key|$$

The more important part is the first term. It make a lot of sense to write such upper-bound in the following parameterized form (X, Y):

$$\min_{0<r<1} \frac{1}{Y} \cdot \frac{1}{1-r} \cdot \mathrm{Opt}(X \cdot r \cdot C, I)$$

In our case, $X = 2/3$ and $Y = c/(1+c)$. The most direct reason is that, this form is tightly related to the (h, k) paging problem in the study of page-replacement algorithms. It compare the page miss rage of an on-line page replacement algorithm with cache size k, and the page miss rate of an off-line page replacement algorithm with cache size h. It is said to be $c(h,k)$-competitive ratio if the page miss rate of the on-line algorithm is at most $c(h,k)$ times of the optimal algorithm.

Many algorithms have $\frac{k}{k-h+1}$-competitive ratio, and each of them can be easily transformed to our algorithm with $X = 1$ and $Y = 1$. A list of them includes[46]:

1. Least recetnly used (LRU)
2. First-in, first-out (FIFO)
3. Not recently used (NRU)
4. Gready Dual
5. Flush when full[3]
6. LANDLORD

This is not a coincidence. In fact, $\frac{k}{k-h+1}$ is the best possible for any deterministic online algorithm (i.e. not knowing the future input)(Theorem-5 from [47]). Using exactly the same construction of input sequence, we can obtain equivalent result for reduce-by-key operation and conclude that $(X, Y) = (1, 1)$ is the best what we can get.

So a question is, is our current $(X, Y) = (2/3, c/(1 + c))$ good enough? It is easy to see that both X and Y are the bigger, the better, as $Opt(X, I)$ is a monotone decreasing function of X. The current (X, Y) is close to $(1, 1)$, but still have some gap. In exchange, it ensured a continuous memory access pattern to off-chip memory. Any reduce-by-key algorithm that attempt to achive smaller traffic using a page-replacement algorithm to manage on-chip memory will inevitability have random access since all page-replacement algorithm will ensure there is only one copy of page instead of allowing many copy of unmerged partial sums. As we analyzed earlier, no redundancy will enforce random access (and also write-conflict).

[3]This algorithm is some what similar to our algorithm in a very important aspect that dump all content (as Rule 2a). But there are also differences like we do not "immediate load missing page" for each missing key, but perform multi-way merge in the end. This is not possible in the context of page-replacement, and it is used here because we have an extra purpose about making off-chip memory access not random.

4.6.3 Multi-level Memory Hierachy

We consider the memory traffic across every cross-section of a multi-layer memory hierarchy. The overall analysis method is similar to the two-level case since our algorithm is similar at different memory levels. We highlight the main difference here.

1. In the multi-level case, aggregation operation can still be performed on the top memory level. Therefore the multi-way merge at lower levels must go across multiple cross-sections. Therefore, the traffic between two adjacent levels $lv \leftrightarrow lv+1$ includes the following:

   (a) The dump traffic from level lv to level $lv+1$ (same as two-level case).

   (b) The multi-way merge traffic at level $lv+1$ (same as two-level case).

   (c) The multi-way merge traffic for all levels below $(lv+2, lv+3, \cdots)$. This is the new part.

2. The memory level $lv+1$ now also have finite capacity, and multi-way merge will be triggered more often due to Rule 2b.

3. The number of Load from $lv+1$ to lv is no longer symmetric to the number of Store from lv to $lv+1$.

Accurate analysis is rather complicated. We simplify the problem by considering several approximations that are generally true in real hardware:

1. The capacity is usually growing very rapidly when moving down the memory hierarchy. Therefore, compared to the memory capacity at level lv, the sum of the capacity of all memory above lv (i.e. $< lv$) are negligible to lv's capacity. (e.g. the total capacity of $L1$ and $L2$ caches are smaller than the $L3$ cache).

2. The bandwidth is usually dropping very rapidly when moving down the memory hierarchy. Therefore, considering the traffic between a upper cross-section $T_A = i \leftrightarrow i+1$ level, and a lower cross-section $T_B = j \leftrightarrow j+1$ level ($i < j$), then we can always treat

$$T_A + const \times T_B \approx T_A$$

when we analysis the traffic $i \leftrightarrow i+1$, because: it is either correct, or wrong but doent matter anymore.

   (a) $T_A \gg T_B$, T_B is negligible compared to T_A. Then $T_A + const \times T_B \approx T_A$ hold directly.

   (b) If T_B is non-negligible to T_A, then the precise value of the traffic between $i \leftrightarrow i+1$ doesnt matter a lot. The system bottleneck is at $j \leftrightarrow j+1$. We won't get a wrong performance estimation based on the wrong traffic at the wrong T_A.

3. The input sequence is sufficiently long, such that for any cross-section, a constant term of order $O(|Key|)$ or $O(c_{lv})$ can be treate as small. Here c_{lv} is the capacity of memory capacity except the last one.

With above simplification, when we analysis the traffic between $lv \leftrightarrow lv+1$, we can largely ignore the existence of memory level $1, 2, \ldots, lv-1$ (as their capacity are too small) and the existence of level $lv+1, lv+2, \ldots$ (as their merge traffic, although needing to go through $lv \leftrightarrow lv+1$, but can be ignored here).

With those approximations, the traffic between $lv \leftrightarrow lv+1$ in our algorithm can have

following upper bound:

$$\text{Traffic}(lv \leftrightarrow lv+1) \leq \min_{0<r<1} \frac{1}{Y'} \cdot \frac{1}{1-r} \cdot \text{Opt}(X' \cdot r \cdot C, I)$$
$$\text{where } X' = 2/3$$
$$Y' = \min(\frac{c}{c+1}, 1/3)$$

4.6.4 Discussion

We can have some interpretation for the X and Y in the above equations.

1. X represents how effectively an algorithm uses the on-chip memory capacity to filter off-chip traffic.

2. Y represents "what percentage of memory traffic is really necessary".

Now, instead of using Opt to compute the upper-bound of our algorithm's memory traffic, let's use the Opt is also a nature lower bound of any other possible algorithm when using $(X, Y) = (1, 1)$. Several existing methods demonstrate the deficiency by having worse lower bounds due to the reasons we introduced in Section-4.1.2. This can be clearly reflected by their X and Y values.

The impact of privatization: Consider an on-chip buffer with capacity C. If there are P threads running in parallel and divide the buffer into P parts and privatize it, then, with this single design choice, it's traffic $\text{Traffic}(C, I) \geq \text{Opt}(C/P, I)$. I.e., it's $X \leq 1/P$ no matter what algorithm it uses.

The impact of cachelines: Consider an algorithm that relies on the cache mechanism provided by the hardware to bring frequently used keys into the on-chip memory, such as hash tables. If the cacheline size is 64Byte, and the tuple size is 8Byte, then each cacheline can hold 8 tuples. However, if on-chip memory must be managed at cacheline

grainularity, each cacheline can only hold 1 useful tuple if the $|Key|$ is larger than the on-chip memory by multiple times. This force $X \leq 1/8$. In addition, bringup or storing a tuple will always bring up or store a whole cacheline, which means the memory traffic is amplified by 8 times and force $Y \leq 1/8$.

Therefore, even we ignore other non-ideal fact of their implementation, the best possible result of using privatization and cacheline-supported hash tables already make their (X, Y) far worse than 1.

4.7 Evaluation

4.7.1 RTL Implementation

We implement the R-Core in Chisel, an RTL language that provides the same low-level control on the circuit details as Verilog or VHDL. The design is synthesized using both Synopsys Design Compiler (for ASIC) on a 12nm process at 1.42GHz. The function correctness is verified, and the throughput of tuple insertion (when R-Core is used in isolation) is measured using RTL-level simulation with Verilator. The area is 1.20e5 μm^2, and the power is 9.55 mw.

4.7.2 Methodology

We use simulation to numerically compare our proposed SDHA-based method with existing sort and hash-related methods in several aspects: memory footprint, data traffic, resilience to skewness, and throughput. When we model our baseline methods, we assume they have implemented the optimizations found in recent work to represent the state-of-the-art design. During the modeling, some approximation and simplification are only taken when it is optimistic to our baselines, so it is pessimistic to the presented speed-up

of our method. For example, we neglect the overhead of complicated concurrent hash table resizing problems in hash table-related baselines.

Following are the baseline methods we modeled. All of them are normalized to have identical buffer capacity/bandwidth resources (shown in Table-4.2) for a fair comparison. The total number of independent "threads" in all methods in our evaluation is 128 and they run at 1GHz.

- MS (Merge Sort): Merge sort-based method. Our model refers to the design made by Jun et al[48]. The model assumes vectorized sorting networks and merger trees are used.

- RS (Radix Sort): Radix sort-based method. Our model refers to the radix sorter by Liu et al[49]. The model assumes 8-bit (256-radix) partition is used.

- Shared Hash: Use globally shared hash tables at L_3 to aggregate tuples; the local buffers in L_1, L_2 shown in Table-4.2 are bypassed. Each of the four L_3 buffer is implemented as 8 (same as [22]) banks and each bank support one atomic operation for "hash insertion/update" per cycle. Our model refers to the design of Graphicionado [22], Yang et al's design[50], and Absalyamov et al's design[43, 51] for the on-chip, HBM, and DDR4 memory part respectively.

- Private Hash: Independent hash tables are located at the last level of memory and all buffers above it are utilized as cache with 4-way associativity. Buffer resources are partitioned and privately owned by the different threads.

- Partition Hash: Similar to the private hash except that a partition phased is performed as pre-processing. The partitioning phase also refers to the radix sorter by Liu et al[49].

Following the convention of database-related work, we synthetic sufficiently long input stream (10^{10} tuples) with different key cardinality ($|Key|$) using the following four commonly used patterns:

- Uniform: All keys are sampled from a uniform distribution independently. It characterizes applications where the locality of keys is bad, such as k-mers in Genomics.
- Heavy Hitter: Half of the tuples share the same keys, while the rest half has a uniform distribution. It characterizes highly skewed datasets.
- Moving Cluster: Only a small subset of keys are frequent at any moment, while this subset changes slowly over time. It characterizes datasets with good temporal locality. We set the window size to be the square root of $|Key|$.
- Zipf: The frequency follows a power-law distribution. This pattern is typical for real-world sparse matrices and graphs.

4.7.3 Effectiveness of SDHA compression

Figure-4.7 shows the compression ratio of the two methods introduced in Subsection-4.4.2. As a rule of thumb, on-chip SDHA arrays are compressed by about 20%, while those off-chip ones are compressed by at least 30%.

4.7.4 Memory footprint

Figure-4.8 shows the memory footprint for different methods for the uniform pattern. Other patterns make little difference in memory footprint. The merge sort, radix sort, and partition-hash are "in the sky" because their memory footprint is $O(|input|)$. The rest three (shared hash, private hash, and the proposed SDHA) have better $\propto |Key|$ memory footprint, while usually $|Key| \ll |input|$. Among the three, the memory footprint of

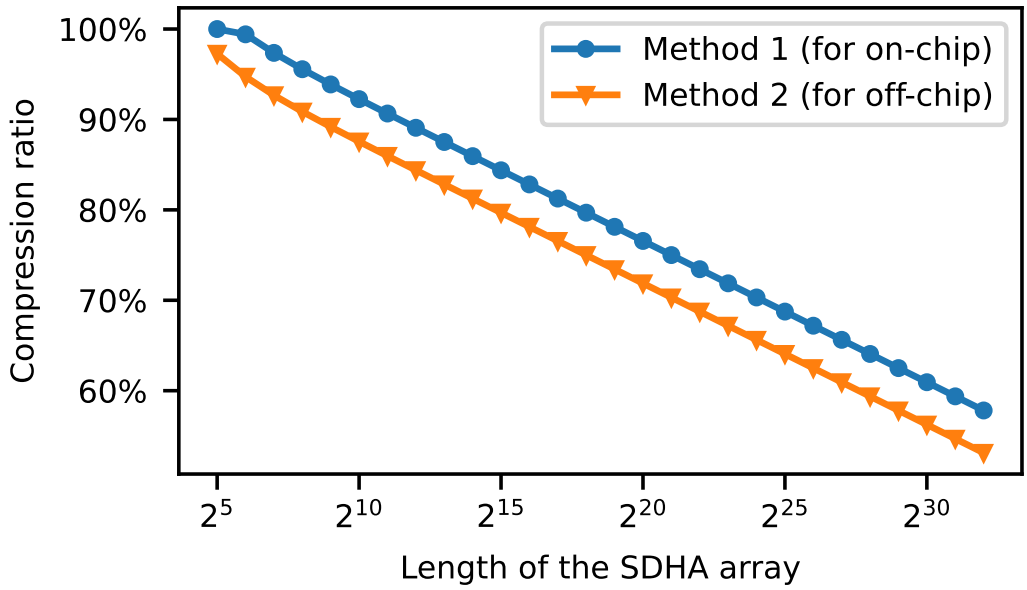


Figure 4.7: Compression ratio of two methods (Method 1 is faster while Method 2 shares a higher compression ratio).

private hash is always $P = 128$ times higher than the shared hash methods due to the duplications of hash tables. Our proposed SDHA spontaneously switches between the two: when the $|Key|$ is very small (especially when $|Key| < P$), it is similar to the private hash method ($\propto P \times |Key|$) – this waste of space here is, in fact, positive because it avoids the contention caused by the phenomenon of "Pigeonhole principle"; while when $|Key|$ grows to be very large, the "Pigeonhole principle" phenomenon diapers, and the memory footprint approaches the shared hash method ($\propto |Key|$) which is more space efficient. This is not a deliberate action but the nature consequent of the Rule 2a,b and 3.

4.7.5 Data traffic and Throughput

We compare the amount of data movement and throughput between different methods. Usually, the throughput is highly correlated with the traffic on the last cross-section

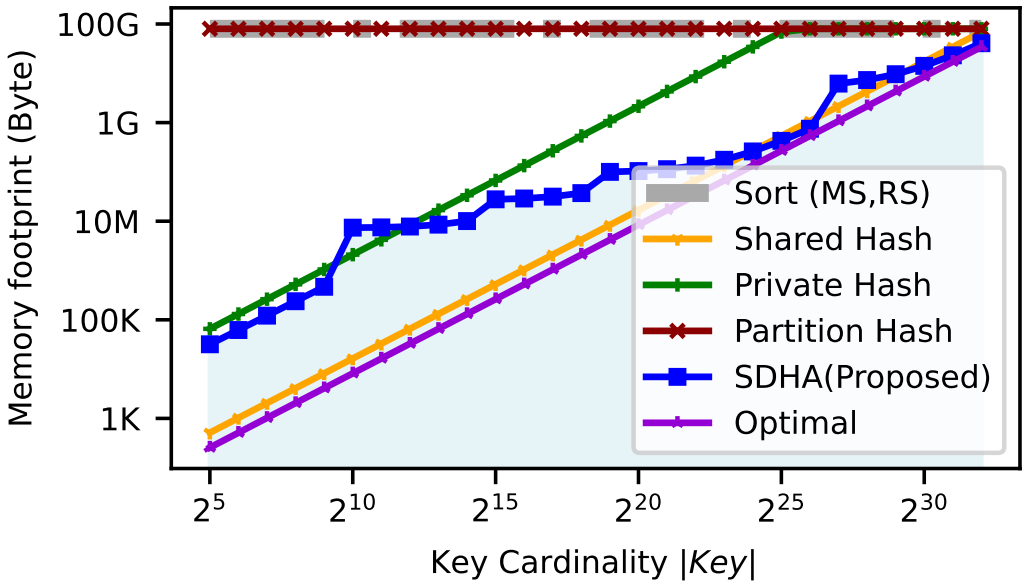


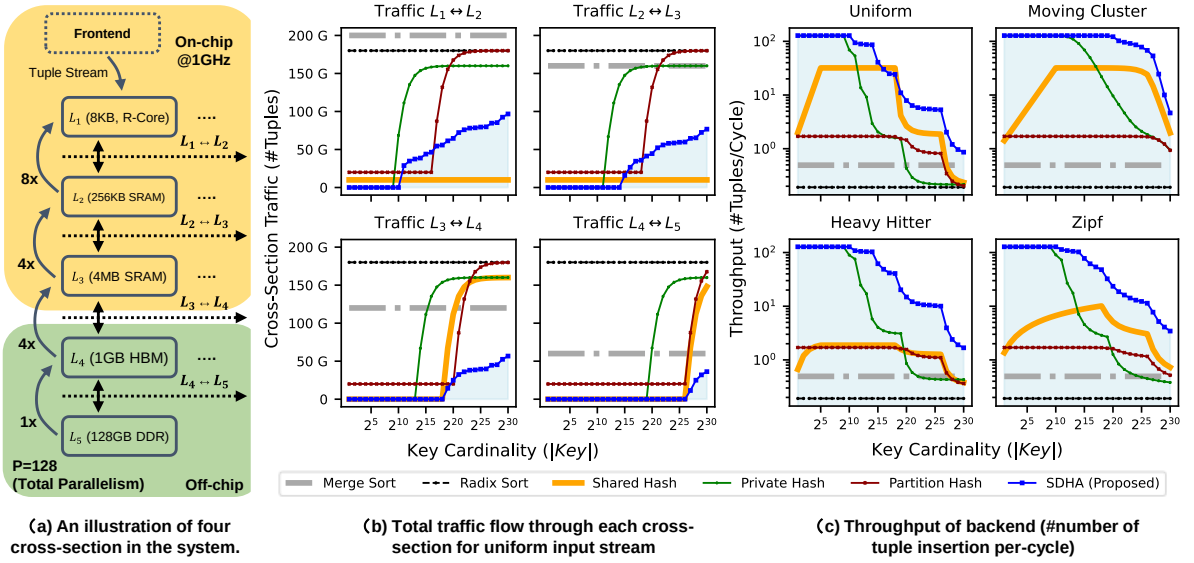
Figure 4.8: The memory footprint for different methods for the uniform pattern.

that has non-negligible traffic, but it will also be influenced by other factors such as contention.

Traffic: Because our system has four cross-sections between five levels of memory hierarchies, as depicted in Figure-4.9-(a), we study the traffic for all of them. In Figure-4.9-(b), we measure the traffic for a uniform input stream with fixed length 10^{10} tuples and different key cardinality. We can observe that:

1. The data traffic for the sort-based method (merge sort and radix sort) is irrelevant to the key cardinality($|Key|$), while the hash-based methods are sensitive to the growth of $|Key|$ especially when $|Key|$ is close to the capacity of the buffer capacity, where a cliff-like increase in traffic can be observed due to the failure of cache. Before the failure point, the data traffic is negligible for the shared/private hash method and SDHA method thanks to the eager aggregation of tuples.

2. The partition hash method has a similar failure point except it has background



(a) An illustration of four cross-section in the system.

(b) Total traffic flow through each cross-section for uniform input stream

(c) Throughput of backend (#number of tuple insertion per-cycle)

Figure 4.9: Traffic and throughput comparison between different methods.

traffic in all cross-sections due to the partition pre-processing without regard to the $|Key|$. This makes partition hash uncompetitive: the SDHA or other hash methods have near zero traffic to the last level memory (DDR4) when $|Key|$ is not too large. Even though it has the advantage in the traffic to the upper levels of memory, the partition method is still much slower because the traffic to the last level of memory will demonstrate the time.

3. Private hash is only competitive when $|Key|$ is small. When $|Key|$ increases, and due to the privatization, its intermediate memory level begins to spill traffic to the next memory level earlier while the SDHA method still has near-zero traffic there. So it always has "the disadvantage of a level of memory hierarchy".

4. The shared hash method is as competitive as the proposed SDHA method in terms of off-chip traffic. Its major limitation, however, is the contention which is not reflected here in traffic and will be discussed later.

5. The proposed SDHA-based method has minimum traffic in almost every case (except in layer L_1 and L_2 which is bypassed by the shared hash method). Compare to the sort-based method, it enjoys early aggregation. Compared to several hash-based methods, the advantage in traffic reduction comes from several factors: 1) more effective utilization of buffer capacity, 2) there is no useless data transmission due to the 64byte cacheline granularity, and 3) SDHA array compression.

A question is whether our baselines, the hash table-based methods, can be significantly improved by using a smaller cacheline to match the SDHA method. The answer is NO and we at best replace the "transmission of unused data" with the "idling cycles of memory bus". The fundamental limitation of the hash tables in this problem is:

- Hash tables demands memory throughput in terms of "number of distinct addresses" instead of just "number of bytes".
- Reduce the cacheline size by 8 times does not automatically increase the system's "addresses throughput" by 8 times, but makes the data bus idle instead (e.g. DDR4 memory).
- It is much harder to double "address throughput" than "byte throughput" in the memory system.

In this aspect, the proposed SDHA-based design will be favored in the long term because it only requires continuous access and only demands "byte throughput".

Throughput We measure the sustainable backend throughput, i.e. how many tuple insertions can be finished in a cycle. In this experiment, we assume the frontend can always generate tuples sufficiently fast to saturate the backend. The tuple generation is application dependent. For example, on SpGEMM, tuples are generated by computing

the outer product of two sparse vectors. The generated tuples are directly forwarded to the reduce-by-key cores(R-Cores) without a round trip to the memory.

Under this setting, figure-4.9-(c) shows the backend throughput of four traffic patterns (Uniform, Moving Cluster, Heavy Hitter, Zipf) for SDHA and baseline methods. We can observe that:

1. When $|Key|$ is very small ($|Key| < 1000$), the SDHA and private hash method have the best performance because the working set fits into the smallest buffer.

2. As $|Key|$ grows, SDHA method consistently performs well while the throughput of private hash drops quickly. One reason for this difference is private hash method privatized the intermediate shared buffers (L_2 and L_3) which reduced their effective capacity, while SDHA did not. The shared hash method takes over to become the second-best method below SDHA.

3. The shared hash method is very sensitive to contention and the performance can drop by more than a magnitude. Contention happens when 1) the key cardinality is very small ($|Key| < 2^5$), as shown in the left end of subfigures in Figure-4.9-(c), or 2) the frequency of different keys is highly non-uniform, such as Heavy Hitter (50% of the tuple share the same key) and Zipf distribution.

4. The partition hash method and two sort-based methods have stable performance irrespective of the input pattern or key cardinality. On the other hand, this insensitivity also makes their performance relatively uncompetitive when $|Key|$ is very small since they cannot exploit this fact. They are mainly bounded by the bandwidth of the last memory level (DDR4) which is only 32GB in our setting.

5. Overall, the SDHA-based method consistently performs well for different traffic patterns and key cardinality. This stability allows it to be used as the "one-for-all"

multi-purpose design for the reduce-by-key backend.

4.8 Conclusion

The reduce-by-key operation is the common backend of important workloads including graphs, genomics, sparse computing and database. Those application scenarios not only expose some common challenges like random access and write conflicts but also impose personalized requirements such as space efficiency, being on-line, and resilience to the skewness. Existing methods cannot simultaneously meet that requirement but have to sacrifice one aspect for another.

This work presents a new method built on top of linear-time construction and multi-way merge of sorted&deduplicated hash array (SDHA); both computations be efficiently finished in the proposed "Reduce-by-key Core" architecture. This design checks all the boxes we desired: no random memory access, write-conflict-free, provably near-optimal memory footprint and data traffic, resilient to skewness, and compression friendly. In our evaluation, the SDHA-based method shows robust performance in all input patterns and all key cardinalities and is steadily one of (or the only) the fastest methods, sometimes even a magnitude better than the second-best choice. This allows the design to become an all-purpose backend design for the reduce-by-key operation.

Chapter 5

Acceleration of Sparse Tensor Decomposition Using PE-Interactive Design

In this chapter, we introduce an accelerator for sparse tensor decomposition (SpTD). Given a set of (key, value) pairs, where the domain of keys results from the combination of several smaller sets, SpTD can factorize these pairs into multiple smaller components. When these components undergo tensor multiplication, they yield an approximation that closely mirrors the original (key, value) pairs. This method proves to be a valuable tool for both unsupervised learning and data completion tasks in multi-dimensional tagged data. Its applications span across various domains, including social network analysis, EDA, and image processing.

5.1 Background

Tensor is the generalization of vectors (i.e., 1D data) and matrices (i.e., 2D data). A d-dimension (termed as mode) tensor of size $n_1 \times n_2 \times \cdots n_d$ can be denoted as $\boldsymbol{\mathcal{X}} \in R^{n_1 \times n_2 \times \cdots n_d}$. If most elements in a tensor are zeros, then the tensor is sparse and

can be stored using only $O(nnz)$ rather than $O(n_1 \times n_2 \times \cdots n_d)$ memory, where nnz is the number of non-zeros. Today, sparse tensors have been widely used to represent real-world data. For example, Amazon reviews can be represented as a 3-mode User-Product-Review sparse tensor; publications make up a 4-mode Author-Title-Journal/Conference-Year sparse tensor; and any incomplete scientific simulation sweeping over d variables forms a d-mode sparse tensor. Despite the superior expressive power of tensors, the traditional numerical algorithms targeted on vectors and matrices (such as principle component analysis (PCA)) are not efficient in handling tensors due to the extra dimensions.

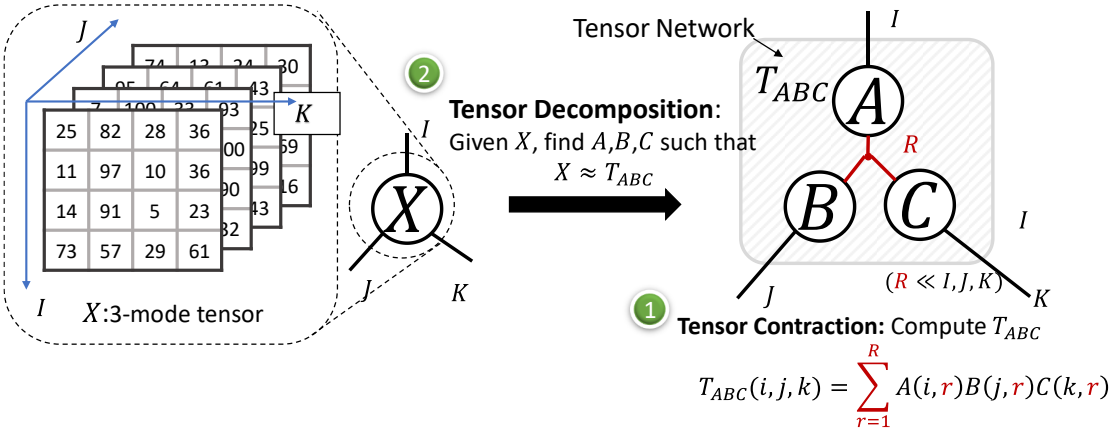


Figure 5.1: Tensor contraction and tensor decomposition.

A prominent solution to address the challenges posed by tensors is the sparse tensor decomposition (SpTD). Given a sparse input tensor, SpTD algorithms typically produce a tensor network that approximates the original tensor using significantly fewer variables, as depicted in Figure 5.1. Prior research has observed that SpTD algorithms are capable of extracting essential data patterns, making them invaluable for big data analysis in fields like social network analysis [52, 53], discussion tracking [54], Internet traffic analysis for cybersecurity [55, 56], and healthcare [57, 58]. Another significant application of SpTD lies in the completion of missing data: a majority of the zero entries in a tensor can be interpreted as missing data, and the tensor network derived from SpTD can function as

a trained model to predict these missing values [59, 60, 61, 56]. Additionally, SpTD can drastically reduce data volume and computational costs [62, 63]. However, despite its vast applicability, efficiently processing SpTD presents significant challenges.

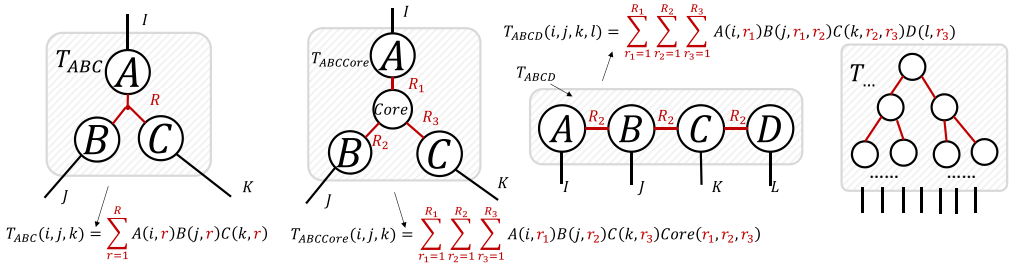


Figure 5.2: Different tensor network structures: CP, Tucker, TT (tensor train), and HT (hierarchical Tucker), from left to right.

A tensor network is a graphical representation used to illustrate the multiplication of multiple tensors, showcasing the structure of tensor decomposition. In this network, each node symbolizes a tensor. The edges emanating from a node correspond to its modes (i.e., dimensions). As an example, a node representing a vector (a 1D tensor) will have exactly one edge, whereas a node representing a matrix (a 2D tensor) will have two edges, and so on. If an edge bridges two (or more) tensors, as depicted by the red lines in Figure 5.1, this indicates that the tensors will be multiplied and aggregated along this mode. An edge not connected to another tensor signifies that the corresponding dimension remains intact. Ultimately, after this matrix multiplication within the entire tensor network, a new tensor is derived. Comprehensive discussions on the concept and applications of tensor networks can be sourced from previous studies like [64]. Taking Figure 5.1 as a case in point, the factor tensors A, B, and C compose a tensor network, with indices i, j, and k retained in T_{ABC}, while r is omitted through summation:

$$T_{ABC}(i,j,k) = \sum_{r=1}^{R} A(i,r)B(j,r)C(k,r). \tag{5.1}$$

In real-world scenarios, tensor networks can manifest in varied structures, as illustrated in Figure 5.2. The method of computing the resulting tensor (i.e., T_{ABC}) for a specified tensor network is termed tensor contraction. In contrast, tensor decomposition describes the reverse process—determining a fitting tensor network such that its contraction approximates a given tensor. This paper emphasizes tensor decomposition. To avert ambiguity, in subsequent sections, we refer to the tensor awaiting approximation as the input tensor (sparse in SpTD contexts), and the tensors within the tensor network as factor tensors. A tensor or matrix is termed low-rank if it emerges as the contraction result of a tensor network, and the contracted edges (dimensions) of this network are minimal. To grasp the essence of SpTD without delving deep into its background, one can examine a rudimentary SpTD instance. For instance, the subsequent equation demonstrates the tensor decomposition challenge linked to the tensor network portrayed in Figure 5.1:

$$\underset{A,B,C}{\arg\min} \sum_{i,j,k} \left(\boldsymbol{\mathcal{X}}_{i,j,k} - \sum_{r=1}^{R} A(i,r)B(j,r)C(k,r) \right)^2 . \tag{5.2}$$

5.2 Challenge and Motivation

As highlighted in Section 5.1, executing SpTD algorithms on traditional general-purpose architectures proves inefficient. In this section, we delve deeper into the challenges, setting the stage for our proposed solution.

5.2.1 Limited Algorithm Flexibility

A pivotal step in accelerating domain-specific applications involves distilling common computation patterns and extracting reusable kernels. However, achieving this in the SpTD domain is challenging due to the inherent algorithmic diversity. Generally, an SpTD algorithm manifests in the form of the subsequent optimization problem:

$$\begin{aligned} &\underset{factor\ tensors}{\arg\min} \sum_{i,j,k\in\Omega} E(\boldsymbol{\mathcal{X}}_{i,j,k},\ \boldsymbol{\mathcal{T}}_{i,j,k}), \\ &\boldsymbol{\mathcal{T}}_{i,j,k} = \text{Contract}(factor\ tensors)_{i,j,k} \\ &\text{s.t.} \quad additional\ constraints \end{aligned} \tag{5.3}$$

Here, $\boldsymbol{\mathcal{T}} = \text{Contract}(factor\ tensors)$ calculates the anticipated tensor through contracting the tensor network, while E measures the approximation error between the predicted tensor $\boldsymbol{\mathcal{T}}$ and the input tensor $\boldsymbol{\mathcal{X}}$. Evidently, SpTD algorithms can exhibit: 1) diverse tensor network structures (embodied by "Contract"), 2) varied loss functions that determine the approximation error (i.e., E), 3) multiple optimization methodologies for deriving solutions, and 4) occasionally, distinct constraint conditions.

Consequently, the kernels necessitated by SpTD algorithms can significantly vary, as illustrated on the left side of Figure 5.3. Even though prior research [65] provides a commendable illustration of the myriad loss functions encountered when deploying the CP tensor network topology combined with the gradient descent optimization technique, the diversity highlighted therein arises from merely altering a subset of the aforementioned factors.

While the prevalent BLAS library addresses some operations necessary for SpTD, such as matrix multiplication and matrix inversion, it is ill-equipped for handling sparse tensors. Recognizing this limitation, specialized kernels designed to optimize the process-

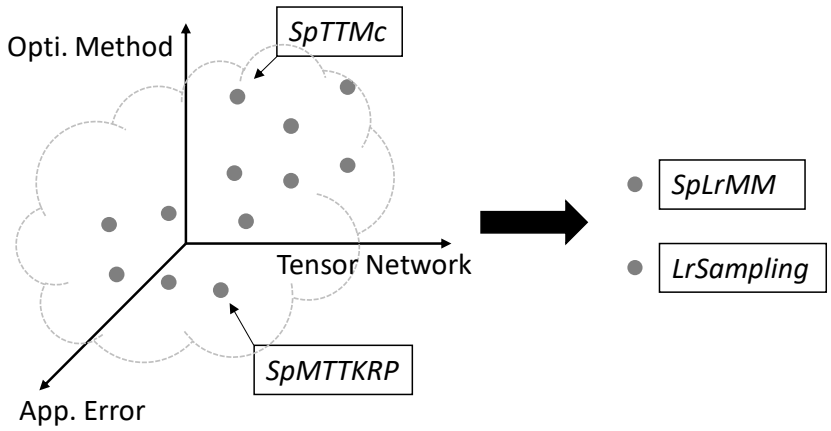


Figure 5.3: Algorithm diversity of SpTD (left) and two proposed core kernels to cover most algorithms (right).

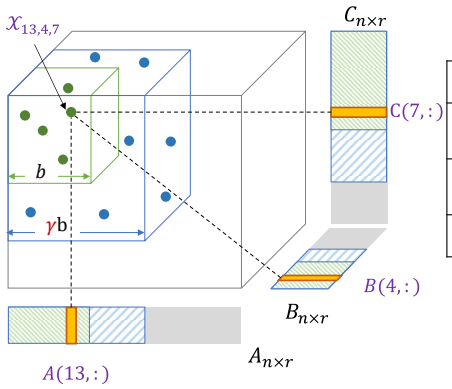
ing of sparse tensors on general processors or distributed platforms have been introduced. Notable among these are the SpMTTKRP kernel [66, 67, 68, 69, 70, 71, 72, 73, 74] and the SpTTMc kernel [66, 68, 75, 72, 73, 76]. Nevertheless, these kernels have their restrictions. Specifically, SpMTTKRP is primarily geared towards CP decomposition with square loss, while SpTTMc is tailored for Tucker decomposition employing the HOSVD or HOOI methods. In essence, their application is not sufficiently broad.

Requirements. To sidestep this piecemeal approach and foster a more encompassing acceleration of SpTD research, it's crucial to discern a unified abstraction for kernels that can accommodate a range of algorithms.

5.2.2 Variable Buffer Size Requirement and Insufficient Data Movement Bandwidth

SpTD algorithms have a distinctive data access pattern. Specifically, when engaging any sparse entry of $\boldsymbol{\mathcal{X}}$, such as $\boldsymbol{\mathcal{X}}_{ijk}$, the data access is confined to a small fragment of factor tensors, namely $A(i,:)$, $B(j,:)$, and $C(k,:)$, which are delineated by the indices i, j, and k. Consider two sparse entries, $\boldsymbol{\mathcal{X}}_{ijk}$ and $\boldsymbol{\mathcal{X}}_{i'j'k'}$. If they share some indices, data

access from factor tensors can be repurposed. For instance, the data from $A(i,:)$ can be reused for $A(i',:)$ if $i = i'$. However, without meticulous preprocessing, the order of indices in consecutive sparse entries can be unpredictable, stemming from their inherent randomness. This unpredictability obscures the potential for data reuse.



Tilling box size	b	γb
Non-zero sparse entries (nnz)	ρb^d	$\rho\gamma^d b^d$
Required Buffer Size	$O(b)$	$O(\gamma b)$
Data Reuse Rate	$O(\rho b^{d-1})$	$O(\rho\gamma^{d-1} b^{d-1})$

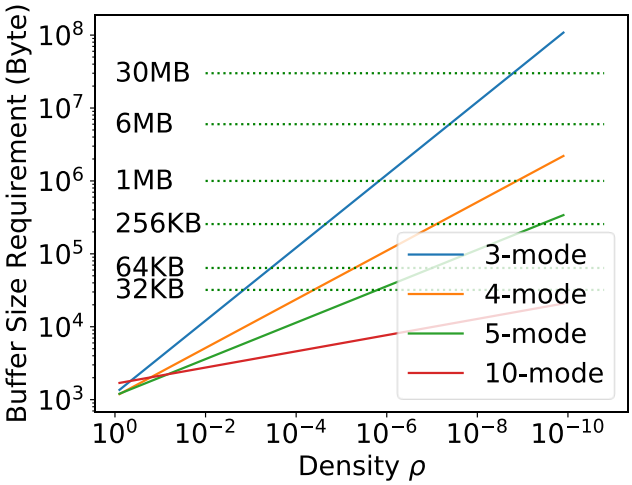
Figure 5.4: Data reuse analysis in SpTD.

To optimize data reuse among sparse entries, it is ideal for adjacent entries to be processed concurrently, as illustrated in Figure 5.4. For instance, by processing the sparse entries in a tiling box where each dimension is b, we optimize efficiency. Given that the density of the sparse tensor $\boldsymbol{\mathcal{X}}$ is ρ, the number of sparse entries within this box is roughly ρb^d. But, due to inter-sparse entry data reuse, only data of size $O(b)$ rather than $O(\rho b^d)$ is required. This optimization necessitates a buffer of size C_{buffer} that can accommodate the data of size $O(b)$, leading to the proportionality $b \propto C_{buffer}$. Given this, we can infer the data reuse rate λ as:

$$\lambda \propto \frac{\rho O(b^d)}{C_{buffer}} \propto \rho O(b^{d-1}) \propto \rho C_{buffer}^{d-1}. \tag{5.4}$$

This property leads to a challenge: when the tensor density ρ decreases (i.e. sparsity increases), the data reuse rate drops in proportion to the decrease in tensor density. Unfortunately, ρ in real-world tensors usually varies over a wide range, such as from 10^{-1}

to 10^{-9}, causing an unbearably low data reuse rate. However, this property also brings an opportunity: if the buffer capacity increases by a factor of γ, the data reuse rate increases by a factor of γ^{d-1}, which can mitigate the negative effect of the decrease of ρ.



		CPU	GPU
Local Buffer	N	28 cores	80 SMs
	C	1056 KB	384 KB
	B	140~360 GB/s	172~2,457 GB/s
	L	4~13 cycles	28 cycles
Global Buffer	C	38.5 MB	6 MB
	B	40.5 GB/s	27 GB/s
	L	50~70 cycles	193 cycles
Off-Chip Mem.	B	5 GB/s	11.25 GB/s

Figure 5.5: Variable buffer size requirement affected by the tensor density ρ. Here we take the CP decomposition with $R = 8$ as an example. The table lists the number of cores/SMs (N), local memory size per core/SM or memory size of global buffer (C), buffer bandwidth (B), and local buffer hit latency (L). The global buffer/off-chip memory bandwidth values are evenly distributed onto each core/SM. The CPU and GPU listed above are Intel Xeon Platinum 8180 and NVIDIA Titan V, respectively.

The unique property of SpTD in Equation (5.4) leads to a variable buffer size requirement based on the sparsity of the input tensor. If we aim for a satisfactory data reuse rate[1], then it varies as illustrated on the left side of Figure 5.5. This can be contrasted with the table on the right that displays the actual buffer capacities and their bandwidths in traditional general-purpose processors. The strategy of "fixed small local buffer & large global buffer" appears inadequate here. Specifically, the needed buffer size

[1]Currently, we use the value of 10 as an example, which is merely an approximation for illustration. However, it does not affect the main conclusion of Figure 5.5. The emphasis in Figure 5.5 is on the slope of the curves. A flatter slope suggests that enlarging the buffer size in SpTD is vital to maintaining a satisfactory data reuse rate. This approximation slightly influences the offset of the y-axis but doesn't affect the slope. The satisfactory data reuse rate in practice depends on multiple factors, such as 1) the intrinsic computation per data when processing one sparse entry, 2) the hardware's compute power to memory bandwidth ratio, and 3) our satisfaction level given the resource utilization.

often surpasses that of the local buffers. Meanwhile, the global buffer, although more sizable, delivers a significantly reduced bandwidth compared to the local ones. Moreover, the local buffers can't be repurposed to extend the global buffer capacity when needed.

Requirements. The aforementioned issues present two primary requirements for architecture design: i) it should facilitate a flexible buffer configuration to adapt to varying tensor sparsity; ii) it must address the limitation of on-chip interconnection bandwidth.

5.2.3 Difficult Kernel Fusion

When tasked with executing multiple kernels, there is often an opportunity to enhance efficiency by merging them. This fusion offers two main advantages: i) data required by several kernels may be loaded a single time; ii) some computation results can be shared among different kernels. However, achieving kernel fusion for parallel execution is challenging due to memory write conflicts arising from distinct kernels. We initially observe this post-fusion (as depicted in Algorithm 11), where every sparse entry in the input tensor (e.g., $\boldsymbol{\mathcal{X}}_{ijk}$) must append results to various objects like $Out_1(i,:)$, $Out_2(j,:)$, and $Out_3(k,:)$. Running multiple sparse entries in parallel could lead to write conflicts, especially when any of the indices i, j, or k overlap among the entries.

Figure 5.6 and 5.7 illustrate the challenges posed by write conflicts when attempting kernel fusion. Consider a hypothetical fused kernel wherein each sparse entry, denoted as $\boldsymbol{\mathcal{X}}_{ijk}$, contributes a partial sum to three outputs: $Out_1(:,i)$, $Out_2(:,j)$, and $Out_3(:,k)$. The two figures depict strategies to avoid write conflicts, both with and without kernel fusion. Both scenarios involve partitioning the sparse entries of the 3-mode input sparse tensor into distinct groups for parallel execution. A sparse entry labeled by the index (i,j,k) can be visualized as a point in a 3D space, with differently colored regions indicating separate PEs.

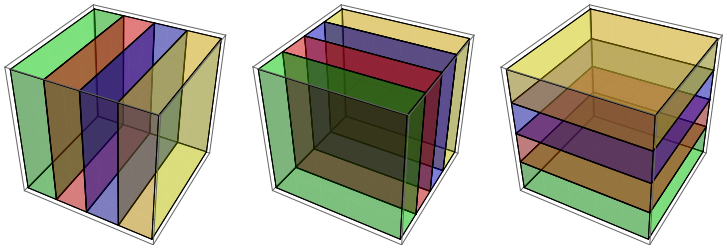


Figure 5.6: In the absence of kernel fusion, kernels are executed in three distinct runs, each producing one of the three outputs. Left: Partitioning of sparse entries based on index i. These entries are subsequently written to $Out_1(:, i)$, allowing write conflicts to be easily circumvented. Middle: Analogous to the left, but partitioning is based on index j, with entries written to $Out_2(:, j)$. Right: Similar to the left, but partitioning is based on index k, directing entries to $Out_3(:, k)$.

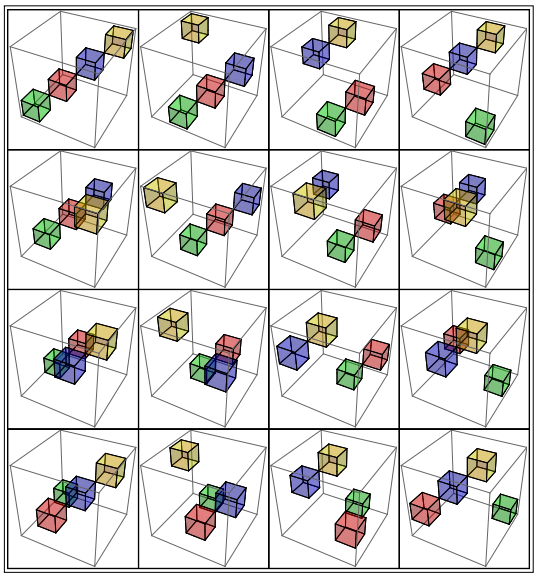


Figure 5.7: When implementing kernel fusion, it becomes necessary to partition the sparse entries based on the indices i, j, and k simultaneously. This approach results in over-partitioning and introduces complexity in scheduling and synchronization tasks.

Several software-level solutions have been proposed to address the write conflict associated with kernel fusion. However, none are entirely satisfactory for our needs. Specifically, the conflict-free partition method [77, 78] segregates the sparse entries across different threads (or PEs) ensuring their indices (e.g., i, j, k) do not overlap. This approach, although conflict-free, leads to over-partitioning of the workload (meaning P^d partitions for P threads) and poses challenges in terms of intricate scheduling and synchronization. Compounding the problem, any conflict-free partition strategy will inadvertently hamper buffer sharing between threads. This sharing is crucial for SpTD, especially given the extensive buffer capacity required to manage highly sparse data.

Requirements. Given the challenges of mitigating write conflicts at the software level, it's imperative to tackle the issue from an architectural perspective. Essentially, the architecture should be capable of handling operations akin to "atomic-add" on a large scale.

5.3 Algorithm Abstraction

To address the issue of algorithmic diversity, we first introduce two general core kernels that are parameterizable based on the tensor network structure. These kernels are termed as SpLrMM and LrSampling. Subsequently, we demonstrate that these kernels are sufficiently versatile to encompass a wide range of SpTD algorithms.

5.3.1 SpLrMM Kernel

SpLrMM stands for sparse-matrix low-rank-matrix multiplication. This kernel is a variant of matrix-matrix multiplication in which one of the two operand matrices is

sparse, while the other is of low rank. It can be succinctly represented as:

$$Out = W^T X \tag{5.5}$$

where X is sparse and W is of low rank. The low-rank matrix W can be represented by a tensor network. Figure 5.8(a) provides an example wherein W comprises three internal factor tensors: A, B, and C. This particular example of the SpLrMM kernel is used in TT-based decomposition. It is primarily chosen for illustration because it encompasses a wider variety of operations. By altering the network structure within W, diverse variants of SpLrMM kernels can be devised to cater to different SpTD algorithms. Both the previously mentioned SpMTTKRP and SpTTMc are merely specific instances of SpLrMM.

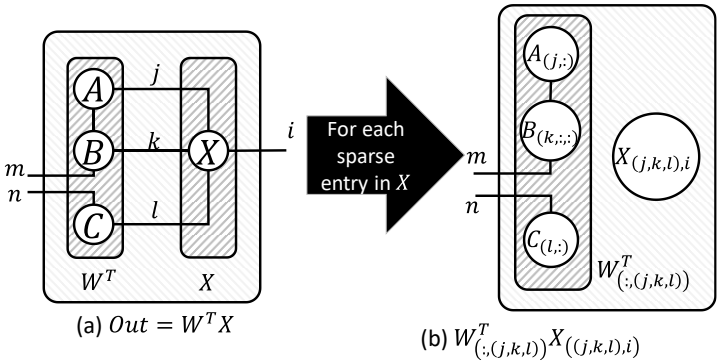


Figure 5.8: An example of performing SpLrMM.

We will exploit two distinct properties of W and X to execute SpLrMM efficiently: the low-rankness of W and the sparsity of X. The former property enables us to compute the result directly from the tensor network representation without the need for an explicit construction of W, while the latter property lets us process only the non-zero entries. To demonstrate how these properties can be leveraged, let's refer back to the example in Figure 5.8(a). Each entry in the sparse matrix X can be treated separately, yielding a

partial sum that is appended to the respective column of Out. This is depicted in Figure 5.8(b) and can be equivalently expressed as:

$$Out_{(:,i)} \leftarrow Out_{(:,i)} + W^T_{(:,(j,k,l))} X_{((j,k,l),i)}. \tag{5.6}$$

For the sparse entry $\boldsymbol{\mathcal{X}} i, j, k, l$, the column $W^T(:, (j, k, l))$ solely depends on $A(j, :)$, $B(k, :, :)$, and $C(l, :)$. The result is subsequently accumulated onto $Out(:, i)$. Fundamentally, all data accesses hinge directly on the indices of the sparse entries. The pseudocode for SpLrMM is outlined in Algorithm 8, which corresponds to Figure 5.8.

Algorithm 8: Pseudocode for SpLrMM

```
for No. ∈ {1, 2, ..., nnz} do
    x ← Value(No.), i, j, k, l ← Index(No.)// x = 𝒳_{i,j,k,l}
    a ← A(j, :), b ← B(k, :, :), c ← C(l, :)
    ------------------------------------------------
    t1 = ax (vector-scalar product), t2 = t1 b (vector-matrix product)
    t3 = c^T t2 (vector outer product)
    ------------------------------------------------
    Out(:, i) ← Out(:, i) + reshape(t3)
end
```

Notice that SpLrMM differs from sparse-dense matrix multiplication, even though it is mathematically correct to regard both as matrix multiplication. Let's use the Email [79] dataset as an example. This sparse tensor has dimensions of $6K \times 6K \times 224K \times 1K$ and contains 54 million sparse entries. To decompose this tensor using TT-based SpTD algorithms (assuming we set $rank = 16$), we must perform a SpLrMM with W being a $(6K \times 6K \times 1K) \times (16 \times 16)$ dense matrix and X being a $(6K \times 6K \times 1K) \times 224K$ sparse matrix (with 54 million non-zero entries). If we naively employ the conventional sparse-dense matrix multiplication method to compute the $W^T X$ operation, we first need to recover W from the tensor network format before executing $W^T X$. This is clearly impractical and inefficient due to W's exponential size (with $9.2E12$ elements). It's essential to utilize the low-rank property of matrix W. In this example, W has an

internal tensor network format as depicted in Figure 5.8, and we only need to store A, B, and C which have smaller sizes of $6K \times 16$, $1K \times 16 \times 16$, and $6K \times 16 \times 16$ respectively (or, with only $9.6E4$, $1.5E6$, and $2.6E5$ elements, respectively). Furthermore, we only compute the columns of $W_{(:,(j,k,l))}$ when required by a sparse entry. Additionally, we can employ the most efficient method to compute $W_{(:,(j,k,l))}$ by summing as early as possible (as illustrated in Algorithm 8). The crux of this method is to harness both the sparsity and low-rankness to minimize computation. This was independently identified under various names (e.g., "avoiding intermediate size explosions" or "factoring") in previous software and hardware works (like the recent Tensaurus [80]) for SpMTTKRP and SpTTMc, both of which are specific instances of SpLrMM. We extend this concept to any W with a low-rank internal tensor network representation.

Table 5.1: Mapping SpTD algorithms onto the proposed SpLrMM and LrSampling kernels. The marked data are low-rank tensors and sparse tensors, respectively.

Loss Function	Iterative Method	Update Formula Derivation	Kernel Mapping
Square Loss: $\|X - WH\|^2$	ALS	$H \leftarrow (\lambda I + W^T W)^{-1} W^T X$ Key Components: $W^T W$, $W^T X$	SpLrMM: $\underline{\underline{W^T X}}$ Tensor Contraction: $W^T W$
	Gradient Descent	$\delta H \leftarrow W^T X - (W^T W) H$ Key Components: $W^T W$, $W^T X$	SpLrMM: $\underline{\underline{W^T X}}$ Tensor Contraction: $W^T W$
	Multiplicative Update	$H_{ai} \leftarrow H_{ai}[(W^T X)_{ai}/(W^T W H)_{ai}]$ Key Components: $W^T W$, $W^T X$	SpLrMM: $\underline{\underline{W^T X}}$ Tensor Contraction: $W^T W$
Masked Square Loss: $\|\Omega_X \circ (X - WH)\|^2$	ALS	$H[:,i] \leftarrow (\lambda I + W_i'^T W_i')^{-1} W_i'^T X[:,i]$ where $W_i' = \text{diag}(\Omega_X[:,i])W$ Key Components (raw): $W_i'^T W_i'$, $W_i'^T X[:,i]$ Key Components (simplified & batched): $(W^T \odot W^T)\Omega_X$, $W^T X$	SpLrMM: $\underline{\underline{(W^T \odot W^T)\Omega_X}}$ SpLrMM: $\underline{\underline{W^T X}}$
	Gradient Descent	$\delta H[:,i] \leftarrow W_i'^T (X[:,i] - W_i' H[:,i])$ where $W_i' = \text{diag}(\Omega_X[:,i])W$ Key Components: $R_1 \leftarrow X - \Omega_X \circ (WH)$, $R_2 \leftarrow W^T R_1$	LrSampling: $R_1 \leftarrow X - \underline{\Omega_X} \circ \underline{\underline{(WH)}}$ SpLrMM: $R_2 \leftarrow \underline{\underline{W^T R_1}}$
	Multiplicative Update	$H_{ai} \leftarrow H_{ai}[(W_i'^T X[:,i])_a/(W_i'^T W_i' H[:,i])_a]$ Key Components (raw): $W_i'^T W_i'$,$W_i'^T X[:,i]$ Key Components (simplified & batched): $(W^T \odot W^T)\Omega_X$, $W^T X$	SpLrMM: $\underline{\underline{(W^T \odot W^T)\Omega_X}}$ SpLrMM: $\underline{\underline{W^T X}}$
KL Divergence: $D_{KL}(X\|WH)$	Multiplicative Update	$H_{a\mu} \leftarrow H_{a\mu}[(\sum_i W_{ia} \frac{X_{i\mu}}{(WH)_{i\mu}})/(\sum_i W_{ia})]$ Decompose into two steps: Step 1: $S_{i\mu} \leftarrow [X_{i\mu}/(WH)_{i\mu}]$ Step 2: $H_{a\mu} \leftarrow H_{a\mu}[(W^T S)_{a\mu}/(W^T \vec{1})_a]$ Key Components: $[X_{i\mu}/(WH)_{i\mu}]$, $W^T S$, $W^T \vec{1}$	LrSampling: $[\underline{X}_{i\mu}/(\underline{\underline{WH}})_{i\mu}]$ SpLrMM: $\underline{\underline{W^T S}}$ Tensor Contraction: $W^T \vec{1}$

Algorithm 9: Pseudocode for LrSampling

```
for No. ∈ {1, 2, ..., nnz} do
    i, j, k ← Index(No.)
    a ← A(i, :), b ← B(j, :, :), c ← C(k, :)
    ------------------------------------------------------------
    t1 = ab (vector-matrix product), t2 = t1 c^T (vector inner product)
    ------------------------------------------------------------
    Out(No.) ← t2 // Out(i, j, k) ← t2
end
```

5.3.2 LrSampling Kernel

LrSampling (sampling elements from a low-rank tensor according to sparse entries). The purpose of this kernel is to sample values (e.g., $\boldsymbol{\mathcal{T}}_{i,j,l}$) from a low-rank tensor $\boldsymbol{\mathcal{T}}$, which is implicitly represented by a tensor network, given a set of sparse entries ($\forall(i,j,k) \in \Omega$). Analogous to SpLrMM, LrSampling is also parameterizable with respect to the tensor network structure. LrSampling can be expressed as:

$$Out = \boldsymbol{\mathcal{T}} \circ \Omega \tag{5.7}$$

where $\circ$ signifies the element-wise product, and Ω contains binary values that indicate the sampling locations. The pseudocode for an example LrSampling operation is provided in Algorithm 9. We observe a consistent access pattern based on the indices of the sparse entries.

5.3.3 Connection to SpTD Algorithms

To demonstrate that the proposed SpLrMM and LrSampling kernels can encompass a range of SpTD algorithms, we initially recast the objective function of a SpTD algorithm into an equivalent matrix representation. For instance, the original objective function, denoted as $argmin_{A,B,C} \|\boldsymbol{\mathcal{X}} - \boldsymbol{\mathcal{T}}_{A,B,C}\|$, can be reformulated as $argmin_{A,B,C} \|X - W_{B,C}H_A\|$. Here, H_A and X are matrices reshaped from A and $\boldsymbol{\mathcal{X}}$, respectively, and the matrix $W_{B,C}$

Algorithm 10: Pseudocode example for CP decomposition without kernel fusion

1) LrSampling Step
for $No. \in \{1, 2, ..., nnz\}$ do
  Load:
  $i, j, k \leftarrow \text{Index}(No.)$// $x = \boldsymbol{\mathcal{X}}_{i,j,k}$
  $a \leftarrow A(i,:)$, $b \leftarrow B(j,:)$, $c \leftarrow C(k,:)$
  Compute:
  $t_1 = b \circ c$ //vector element-wise product
  $t_2 = t_1 a^T$ //vector inner product
  Update:
  $T(No.) \leftarrow t_2$
end

2) An element-wise Step to compute the per-sample gradient for the square loss
for $No. \in \{1, 2, ..., nnz\}$ do
  $G(No.) \leftarrow Value(No.) - T(No.)$ //$G_{i,j,k} = \boldsymbol{\mathcal{X}}_{i,j,k} - T_{i,j,k}$
end

3) SpLrMM Step for A's Gradient
for $No. \in \{1, 2, ..., nnz\}$ do
  Load:
  $g \leftarrow \text{G}(No.)$, $i, j, k \leftarrow \text{Index}(No.)$// $g = G_{i,j,k}$
  $b \leftarrow B(j,:)$, $c \leftarrow C(k,:)$
  Compute:
  $t_1 = bg$ //vector-scalar product
  $t_2 = t_1 \circ c$ //vector element-wise product
  Update:
  $Out_1(:, i) \leftarrow Out_1(:, i) + t_2^T$
end

4) SpLrMM Step for B's Gradient
for $No. \in \{1, 2, ..., nnz\}$ do
  Load:
  $g \leftarrow \text{G}(No.)$, $i, j, k \leftarrow \text{Index}(No.)$// $g = G_{i,j,k}$
  $a \leftarrow A(i,:)$, $c \leftarrow C(k,:)$
  Compute:
  $t_1 = ag$ //vector-scalar product
  $t_2 = t_1 \circ c$ //vector element-wise product
  Update:
  $Out_2(:, i) \leftarrow Out_2(:, i) + t_2^T$
end

5) SpLrMM Step for C's Gradient
for $No. \in \{1, 2, ..., nnz\}$ do
  Load:
  $g \leftarrow \text{G}(No.)$, $i, j, k \leftarrow \text{Index}(No.)$// $g = G_{i,j,k}$
  $a \leftarrow A(i,:)$, $b \leftarrow B(j,:)$
  Compute:
  $t_1 = ag$ //vector-scalar product
  $t_2 = t_1 \circ b$ //vector element-wise product
  Update:
  $Out_3(:, i) \leftarrow Out_3(:, i) + t_2^T$
end

is derived by excluding the A component from the original tensor network $\mathcal{T}_{A,B,C}$. Following this transformation, an update equation for H_A (and, by extension, for A) can be discerned from the matrix representation. Typically, such updates can be efficiently executed using the SpLrMM and LrSampling kernels, as illustrated in Table 5.1.

This procedure can likewise be applied for updating other factor tensors, such as B and C. Herein, the tensor factors undergo iterative updates until convergence is achieved. During each iteration, several SpLrMM/LrSampling kernels are typically invoked, with each kernel constituting a loop that iterates over all sparse entries (as seen in Algorithm 8 and 9). Algorithm 10 provides an example of utilizing gradient descent to accomplish CP-based SpTD using the SpLrMM/LrSampling kernels. Notably, only the second step, a straightforward element-wise subtraction with minimal computational overhead, deviates from the SpLrMM and LrSampling methodologies. However, this deviation is inconsequential and will be seamlessly integrated once all kernels are fused within the unified framework to be discussed. The precise time distribution across different operations is contingent on various hyper-parameters, such as rank and sparsity.

5.3.4 Kernel Fusion and the Unified Framework

It's feasible to merge multiple discrete kernels (or loops) into a singular, cohesive loop to minimize repetitive computation and data access. This can be observed in Algorithm 11[2]. From the preceding discussions, it's evident that the kernels we have proposed are versatile enough to cater to a broad spectrum of SpTD algorithms. Moreover, they are crafted in standard formats that align seamlessly with our designated architecture.

Comparing with the SF3 Framework in Tensaurus [80]. Recent work on Tensaurus [80] introduces an accelerator design tailored for generalized sparse-dense operations.

[2]In certain algorithms, particularly those based on the ALS method, factor tensors necessitate alternating updates. This renders the complete fusion of all kernels unattainable. However, substantial benefits can still be reaped by merging kernels that are designated for updating the same factor tensor.

Algorithm 11: Pseudocode example with kernel fusion

for $No. \in \{1, 2, ..., nnz\}$ do
  //Same function as Algorithm 10, but with kernel fusion enabled.
  Load:
  //All loaded data are reused by LrSampling, SpLrMM-A,B,C
  $x \leftarrow \text{Value}(No.)$, $i, j, k \leftarrow \text{Index}(No.)$// $x = \boldsymbol{\mathcal{X}}_{i,j,k}$
  $a \leftarrow A(i,:)$, $b \leftarrow B(j,:)$, $c \leftarrow C(k,:)$

  Compute:
  $t_1 = b \circ c$//Result reused by LrSampling(t_2) and SpLrMM-A
  $t_2 = t_1 a^T$
  $g = x - t_2$//g is immediately reused, without round trip to DRAM.
  $t_3 = t_1 g$
  $t_4 = ag$//Result reused by SpLrMM-B, SpLrMM-C
  $t_5 = t_4 \circ c$
  $t_6 = t_4 \circ b$

  Update:
  $Out_1(:, i) \leftarrow Out_1(:, i) + t_3^T$
  $Out_2(:, j) \leftarrow Out_2(:, j) + t_5^T$
  $Out_3(:, k) \leftarrow Out_3(:, k) + t_6^T$
end

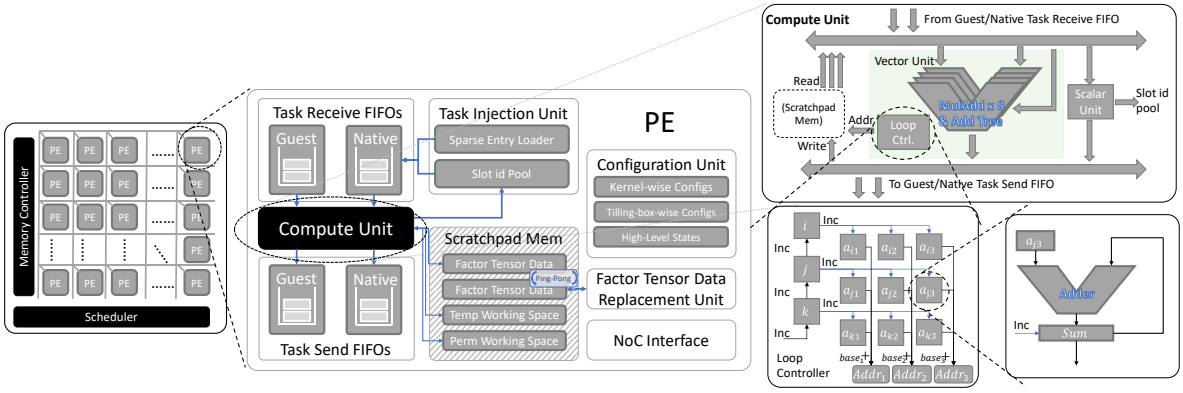


Figure 5.9: Overview of the STE architecture design.

Applications are mapped to Tensaurus using the SF3 framework. Succinctly, the SF3 framework encapsulates operations as:

$$fiber_{out} = \sum_{D1} fiber_1 \ (op) \sum_{D2} scalar \cdot fiber_2$$

where (op) can be a Hadamard product, a Kronecker product of two vectors, or an outer product. Both our framework (as presented in Algorithm 11) and SF3 share similarities, such as supporting SpMTTKRP and SpTTMc. The fundamental distinction between SF3 and STE rests in the order of processing sparse entries. While SF3 navigates all sparse entries in the dictionary order of their indices (i, j, k), STE approaches them based on tiling boxes. SF3's methodology benefits from allowing sparse entries with identical index $D1$ to share computational aspects. This is especially advantageous when the compaction ratio of $D2$ is substantial and the second operation (associated with $fiber_1$ and shared) outweighs the first (linked with $fiber_2$ and not shared) in terms of expense. Conversely, STE leverages kernel fusion for computational efficiency. It's pivotal to highlight that SF3 solely endorses one output target $(Out_1(:, i))$, which hinders Tensaurus from incorporating kernel fusion. This is because kernel fusion necessitates multiple output targets for each sparse entry $(Out_1(:, i)$, $Out_2(:, j)$, $Out_3(:, k)$, and so on). Differences also arise in the memory access patterns of STE and SF3. As previously mentioned, SF3 adheres to dictionary order for accessing sparse entries. This leads to swift changes in index k, causing data like $C(:, k)$ to experience suboptimal data locality and reuse. STE presents a more favorable approach by segmenting the sparse tensor into cubes. Lastly, SF3 poses limitations regarding the maximum supported tensor modes, typically accommodating up to 3-mode tensors. In contrast, our work remains unbounded in this respect.

5.4 STE Architecture Design

In this section, we introduce the STE, an efficient architecture designed to execute the core kernels previously discussed. Figure 5.9 provides an overview of our design, which includes 16×16 PEs, a scheduler, and a memory controller. PEs can communicate via a Network-on-Chip (NoC) infrastructure, forming PE groups for task processing. We detail the design of the major components, outline the task execution flow, and explain the deployment of SpTD algorithms.

5.4.1 Design Philosophy

A central challenge we address is the need for flexible buffer sizes. Each PE possesses a fixed 128KB private buffer. When a task requires a buffer size beyond this limit, PEs are grouped collaboratively, pooling their buffer capacities. The cumulative buffer size of such a PE group is $PE_{Num} \times 128KB$, where PE_{Num} represents the number of PEs in the group. However, merely grouping PEs doesn't fully address the challenge. Fetching data from another PE over the NoC, which has higher latency and lower bandwidth, is considerably slower than accessing the local buffer of a PE.

To mitigate this, we advocate for the shared computational resources of PEs within a group. This involves the active relocation of tasks from one PE to another that already holds the required data, significantly reducing traffic on the NoC. For instance, in matrix-vector multiplication, if a matrix resides in one PE and the vector in another, the task is moved to the PE containing the matrix, necessitating only the movement of the vector. This approach ensures that computations benefit from both the high bandwidth of intra-PE buffers and the extensive capacity of inter-PE groupings.

Secondly, consider the challenge posed by write conflicts in kernel fusion. This necessitates support for operations akin to atomic add, but with high throughput. Our

PE-interactive design addresses this inherently. Each PE retains its private local buffer, and other PEs can only modify this buffer by transferring tasks to the owning PE. As a result, write conflicts are effectively eliminated.

Finally, several other challenges must be addressed to realize the practicality of the aforementioned design. This includes minimizing the communication and control overhead associated with task transfers, ensuring efficient allocation of workspace for tasks given constrained on-chip memory, and avoiding PE idleness and deadlock during task exchanges. These issues have been tackled in our STE architecture, the specifics of which are elaborated upon in the subsequent sections.

5.4.2 Processing Element Specialization

As depicted on the right side of Figure 5.9, a Processing Element (PE) comprises several components: a configuration unit, task receive/send FIFOs, a task injection unit, a factor data replacement unit, a scratchpad memory, a compute unit, and a NoC interface.

Terminology definitions. As tasks traverse the NoC, they interact with multiple PEs. The originating PE is termed the "home PE", while all others are referred to as "remote PEs". Tasks within their home PE are denoted "native tasks", whereas tasks in any other PE are classified as "guest tasks". It's important to note that while a task may deem a PE remote, another task could simultaneously regard that same PE as its home.

Configuration Unit. This unit oversees the high-level states of the PE. It exposes configuration registers and an instruction buffer to the scheduler.

Task FIFOs. Task FIFOs are designed in pairs, with one dedicated to native tasks and the other to guest tasks. As tasks arrive, they are directed to the appropriate receive FIFO based on their status. After undergoing processing in the compute unit, tasks are transitioned to their respective send FIFO (either native or guest). The distinction be-

tween native and guest FIFOs is purposeful, primarily to preempt and prevent deadlocks in the event of congestion, a scenario elaborated upon in subsequent sections. These FIFOs facilitate communication between PEs through message exchanges.

Task Injection Unit. This unit is responsible for fetching sparse entries in COO (coordinate list) format from off-chip memory. For each sparse entry retrieved, it generates a task and directs this task into the native task receive FIFO. A vital feature of the task injection unit is its self-throttling capability. This ensures that the unit doesn't indiscriminately inject new tasks without accounting for the completion status of prior tasks. To achieve this self-throttling behavior, a mechanism employing a slot id pool is in place. Additionally, there's a set limit on the number of active tasks, or "on-the-fly" tasks, that can be initiated by each PE. Whenever a task is introduced, it's allocated a slot id from the pool. This slot id is later returned to the injection unit upon the task's termination (indicating that the task's final phase must be managed by its home PE). The pool initially houses a predefined number of slot ids, symbolized by $SlotMax$ (the aforementioned bound). If the pool is depleted, the injection of new tasks is temporarily halted.

Scratchpad Memory. The scratchpad memory in each PE stores a portion of the factor tensor data and also houses intermediate results as a working space. This working space is categorized into two types: temporary (Temp) and permanent (Perm). Both native and guest tasks can utilize the Temp working space. Once a task departs from this PE, subsequent tasks can overwrite the data in the Temp working space. On the other hand, only native tasks of the current PE can access the Perm working space, requiring a slot id for data retrieval. This necessitates its division into $SlotMax$ separate segments. Each on-the-fly task possesses a secure storage area in its home PE's Perm working space, ensuring data retention even if the task departs its home PE. This data remains intact until the task completes, at which point the slot id is relinquished to the

task injection unit. The presence of the Perm working space reduces data transit in the NoC, as only necessary data is transmitted while the rest stays in the home PE.

In our design, each PE's scratchpad memory has a capacity of 128KB, which contributes to the total on-chip memory of 32MB. The access latency for this memory is 1-cycle. This capacity adequately accommodates both the temporary and permanent working spaces required by SpTD algorithms with a reasonable rank. However, for unavoidable situations where a very large rank is present, software/algorithm-level partitioning can be explored to prevent overflow. Two possible strategies include distributing the data across different PEs within the same PE group or spreading the computation over multiple runs.

Factor Tensor Data Replacement Unit. The factor tensor data replacement unit prefetches the factor tensor data for the upcoming tiling box into a ping-pong buffer. This mechanism effectively decouples off-chip access from on-chip computation. It's important to note that each PE in a group loads only its designated portion of the factor tensor data.

Compute Unit. This unit supports five types of instructions: i) retrieving a message from task receive FIFOs and directing it to the appropriate PC (program counter); ii) determining which PE possesses the required factor tensor data (e.g., for $A(i,:)$, the result is $mod(i, PE_{NUM})$); iii) executing arithmetic operations; iv) constructing a message and placing it into task send FIFOs; v) relaying the slot id back to the task injection unit upon task completion.

The available arithmetic operations encompass scalar ones (e.g., $+$, $-$, $\times$, $\div$) and tensor ones (e.g., inner product ($\cdot$), outer product ($\otimes$), element-wise product ($\circ$), matrix-vector product). These are designed to support the operations stipulated by SpLrMM (Algorithm 8) and LrSampling (Algorithm 9). We have adopted two techniques to bolster throughput, as illustrated on the right side of Figure 5.9. Initially, these operations

necessitate only a single instruction to execute all tasks, obviating the need for software loops. This instruction, when executed, expands in hardware to initiate a series of micro-operations, thus evading potential instruction overhead. Subsequently, an 8-lane vector unit is utilized for these operations to leverage data-level parallelism. Notably, all the tensor arithmetic instructions mentioned above boil down to one operation distinguished merely by different parameters, as elucidated in Figure 5.10. To implement this, we employ a "loop-controller" (depicted at the bottom right of Figure 5.9) to yield three addresses: $addr1$, $addr2$, and $addr3$ using an "address matrix ($a_{i1} \sim a_{k3}$)". A minor discrepancy exists between the code shown in Figure 5.10 and the hardware depicted in Figure 5.9: we've opted for adders over multipliers in address computation to curtail hardware expenses.

```
for(int i=0;i<i_max;++i){
 for(int j=0;j<j_max;++j){
   for(int k=0;k<k_max;++k){
    int addr1 = base1 + ai1*i + aj1*j + ak1*k;
    int addr2 = base2 + ai2*i + aj2*j + ak2*k;
    int addr3 = base3 + ai3*i + aj3*j + ak3*k;
    mem[addr3] += mem[addr1] * mem[addr2];
   }
 }
}
```

Figure 5.10: Code explaining instructions for inner product ($\cdot$), outer product ($\otimes$), element-wise product ($\circ$), and matrix-vector product are actually only one operation, because all of them can be mapped into the above form.

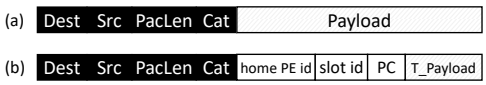


Figure 5.11: Packet format of task messages: (a) general format; (b) specific format for inter-PE communication. The black segments will be processed by the NoC interface.

NoC Interface. The NoC interface manages the receipt and dispatch of packets to and from other PEs. The format of these packets is depicted in Figure 5.11(a). They

consist of the following fields: destination (Dest), source (Src), packet length (PacLen), traffic category (Cat), and Payload. Different traffic categories are designated for packets exchanged between PE↔PE, PE↔scheduler, and PE↔memory controller.

For the communication specifically between PE↔PE, the packet format is detailed in Figure 5.11(b). The header for the Payload is streamlined, consisting of: the home PE id (8 bits), slot id (8 bits), and PC (16 bits). Here, PC signifies the progression of the given task. The T_Payload transports the necessary data for the destination PE, and both its length and content are subject to variation across different processing stages.

5.4.3 Wrapped Memory Controller

The memory controller is encapsulated within a PE-like structure and termed as a "Reduction PE" before being integrated into the PE array. The components within the "Reduction PE" are largely similar to those in a standard PE, except that the 128KB local buffer is replaced by an off-chip memory controller for the DRAM interface. The motivation behind this design is to address the potential write conflicts. Even though write conflicts within each PE group are effectively mitigated through the compute transfer, they might still arise between PE groups when their index ranges overlap (e.g., when both PE groups cover the range from i to i+31). By configuring the memory controller as a PE, this inter-PE group write conflict is avoided. After a PE group completes a tiling box operation, the standard PE's factor data tensor replacement unit dispatches the final partial sum of factor tensors (e.g., $Out(:, i : i + 31)$) from the current tiling box to the "Reduction PE" in a manner akin to standard task delegation. Subsequently, this "Reduction PE" accumulates the partial results directly into the DRAM. Since the "Reduction PE" utilizes a task receive queue, the accumulation of partial results from various PE groups occurs sequentially, ensuring the absence of write conflicts.

The 'Reduction PE' doesn't require high throughput; employing 8 adders[3] as seen in standard PEs suffices. This is because most reductions are predominantly executed within each PE group, and the "Reduction PE" is primarily responsible for the ultimate reduction of partial sums from distinct PE groups.

5.4.4 The Lifetime of A Task

From the core kernels outlined in Algorithm 8 and 9, it's apparent that each sparse entry necessitates several operations as itemized in the loop body. For every sparse entry, we instantiate a task, culminating in a total of nnz tasks for each kernel execution. Given the lightweight nature of each task, the large number of non-zeros won't contribute significantly to control overhead. Now, let's delineate the lifetime of a task using a streamlined example derived from Algorithm 8. Suppose the sparse entry is $\boldsymbol{\mathcal{X}}_{4,7,2,5} = 3.5$ and the requisite operations for this entry are:

1. Multiply $\boldsymbol{\mathcal{X}}_{4,7,2,5}$ by vector $A(4,:)$ to yield vector t_1.
2. Multiply vector t_1 with matrix $B(7,:,:)$ to produce vector t_2.
3. Calculate the outer product of vector t_2 and vector $C(2,:)$, resulting in matrix t_3.
4. Reshape and accumulate matrix t_3 onto $Out(:,5)$.

It's imperative to note that $A(4,:)$, $B(7,:,:)$, $C(2,:)$, and $Out(:,5)$ could reside in disparate PEs within a group. For simplicity, consider that $A(4,:)$ is located in the PE identified by $id = mod(4, PE_{Num})$. The positioning for $B(7,:,:)$, $C(2,:)$, and $Out(:,5)$ follows a similar logic. Thus, the task embarks on a journey across several PEs to reach completion.

[3]It should be noted that even for multiplicative updates, the primary reduction operation remains addition rather than multiplication. The multiplication with the old factor tensor takes place once all partial results are consolidated.

Initially, an arbitrary PE within the group (e.g., PE0) retrieves the integer indices 4,7,2,5 alongside its value 3.5, subsequently initiating a task. This act designates PE0 as the home PE for the ensuing task. The Task Injection Unit then assigns a *slot id* (say, 3) to the task and lodges it into the Native Task Receive FIFO in the form of a message. This format is illustrated in Figure 5.11 with an initial PC value of 0. The Compute Unit cyclically fetches tasks from the Task Receive FIFOs for execution. As a task gets retrieved and executed, it commences at the instruction delineated by PC within the message (with instruction lists being consistent across all PEs). The execution process persists until a "send task" instruction is encountered. This prompts the generation of a message in the Task Send FIFOs to dispatch the task and cues the Compute Unit to embark on the next task in the Task Receive FIFOs. Drawing from the aforementioned example, the task will launch at PE0 and will undergo four phases:

1. It transitions to the PE containing $A(4,:)$, performs requisite operations, and returns to PE0.
2. It migrates to the PE hosting $B(7,:,:)$, conducts operations, and circles back to PE0.
3. The same sequence ensues for $C(2,:)$.
4. Likewise for $Out(:,5)$.

Upon its final return to PE0, the task restores the *slot id* to the Task Injection Unit and concludes.

5.4.5 Deadlock Avoidance

Congestion might occur when certain PEs turn into hot spots. Overflowing receive FIFOs in these PEs could propagate traffic congestion throughout the NoC, further

obstructing other traffic flows. We observe that, while a PE could potentially have up to $(PE_{Num} - 1) \times SlotMax$ incoming guest tasks simultaneously in the worst-case scenario, it never holds more than $1 \times SlotMax$ native tasks at any given time. This observation leads us to implement the following measures to mitigate deadlock:

1. Tasks must alternately visit their home PE and remote PEs, following a sequence like: home PE→remote PE→home PE→remote PE→...;

2. To ensure native task receive FIFOs don't overflow, the FIFO capacity should be set to at least $Size(packet) \times SlotMax$;

3. Traffic heading out from the home PE to a remote PE should never hinder traffic moving from the remote PE back to the home PE, even if the outgoing traffic is stalled. This can be achieved by dedicating a set of virtual channels in the NoC specifically for the returning traffic.

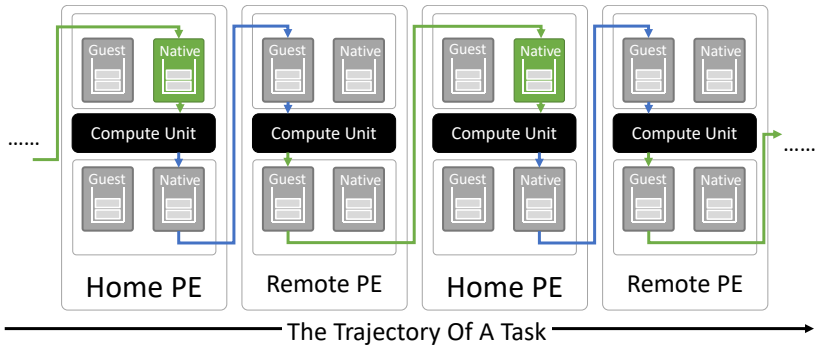


Figure 5.12: The task trajectory across PEs.

By adhering to the aforementioned measures, deadlocks are effectively avoided. As depicted in Figure 5.12, since the native task receive FIFO is guaranteed not to overflow or encounter blockages, traffic from a remote PE returning to the home PE (represented by the green arrows in Figure 5.12) will remain uninterrupted. This ensures the guest send

FIFO can continually dispatch packets, implying that the compute unit can process guest tasks without halt. This in turn signifies that the guest receive FIFO won't experience blockages. Consequently, traffic from the home PE directed at remote PEs (illustrated by the blue arrows in Figure 5.12) proceeds without hitches. To sum up, the deadlock situation is effectively averted.

5.4.6 Mapping Algorithms onto STE

In this section, we delve into the mapping of SpTD algorithms onto STE. This process encompasses algorithm compilation, data preprocessing, and hardware execution, as depicted in Figure 5.13.

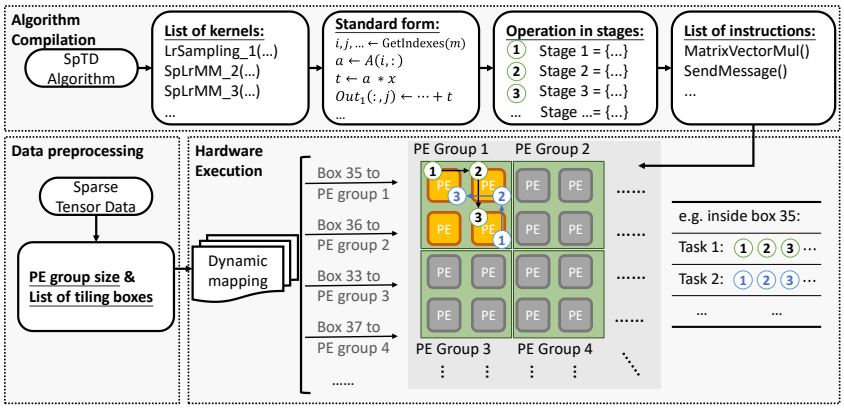


Figure 5.13: Algorithm mapping flow.

Algorithm Compilation. Our process begins with the SpTD algorithm, which we express using the SpLrMM and LrSampling kernels. Subsequently, we integrate these kernels into a standardized format, as illustrated in Algorithm 11. We then categorize the operations from this standard format into multiple processing stages. In this setup, each task undergoes execution for one stage on a singular PE. Intermediate results traverse between stages either through the Perm working space or via the T_Payload field of the task. Preferably, the use of the latter is minimized to curtail on-chip communication

overheads.

Our initial step involves assigning factor tensor data access operations, such as $a \leftarrow A(i,:)$ and $b \leftarrow B(j,:,:)$, to distinct stages. Following this, we allocate compute operations, like $t \leftarrow a * b$, to the most related stage, thereby minimizing data movement. For instance, if $Size(a) < Size(b)$, the operation $t \leftarrow a * b$ is assigned to the stage containing $b \leftarrow B(j,:,:)$, subsequently transferring a over the NoC. In situations where $Size(a) = Size(b)$, either assignment is viable.

Once operations corresponding to a sparse entry are designated to their pertinent stages, we introduce instructions related to "send message" between each pair of stages. This includes determining the subsequent PE id and packaging data into the message payload. We finalize by presenting the program as an instruction set. It is crucial to note that the program for all tasks must be uniform, limiting the compiler to general assumptions. Hence, certain optimizations, specific to particular sparse entries, remain unexplored. A common misconception might be the unlikely occurrence of a non-zero $X_{i,j,k}$, wherein the home PE and remote PEs inadvertently coincide. In such cases, one might assume the compiler can consolidate all steps into a singular stage. However, our design doesn't leverage this potential, ensuring consistent kernel size and compilation time. Presently, our compilation process is manual, but it inherently possesses the capability for automation.

Data Preprocessing. The input sparse tensor is segmented into multiple boxes, each of the dimensions $b_1 \times b_2 \times \cdots \times b_d$. Each of these tiling boxes is then processed within a PE group. The determination of $\{b_i\}$ adheres to two primary guidelines, as detailed in Section 5.2: i) When the tensor density, ρ, is low, the size of the tiling box is augmented to enhance the data reuse rate; ii) Without contradicting the first guideline, the size of the tiling box should be minimized whenever feasible to decrease the necessary buffer size. This in turn results in a smaller PE group, leading to reduced on-chip traffic. With

a designated tiling box size, the size of the PE group is ascertained, and the sparse entries within each tiling box are consolidated into a list of tasks.

Hardware Execution. The scheduler dynamically assigns tiling boxes to PE groups. Although these PE groups operate autonomously, PEs within a group are interdependent and communicate among themselves. Before the onset of computation, all PEs must load the essential factor tensor data into their scratchpad memory. Fortunately, the latency of this loading process can be efficiently masked utilizing a ping-pong buffer strategy. Subsequently, the PEs embark on processing the sparse entries. Upon the completion of all sparse entries in a tiling box, the scheduler then transitions the PE group to the subsequent tiling box.

5.4.7 Minor Optimizations

Relative Indexing. After tensor tiling, the sparse entries only need to store the relative indices within each tiling box. This approach minimizes index overhead and further reduces off-chip memory accesses.

Continuous Tiling Box Assignment. The scheduler always strives to assign adjacent tiling boxes to the same PE group. By doing so, only one piece of the factor tensor data (e.g., $A(i,:)$) needs replacement, while the remaining $d-1$ pieces can be reused consecutively (e.g., $B(j,:)$, $C(k,:)$). This strategy also contributes to a decrease in off-chip memory accesses. The most straightforward method is to schedule tiling boxes in the (i,j,k) dictionary order. A snake-like trajectory can also be effective. It's important to note that ensuring continuity most of the time is sufficient; occasional deviations from this requirement only marginally impact performance.

Random Permutation. In order to achieve a balanced workload distribution among PEs, we implement three additional preprocessing steps: i) Prior to tiling, we randomly

permute the indices of each tensor mode; ii) Post-tiling, we evenly distribute the sparse entries of each tiling box among intra-group PEs; iii) Finally, within each PE, the order of the assigned sparse entries is randomized.

5.4.8 Advantage Summary

Major Advantages.

1. Algorithm Generality: STE supports a broad spectrum of SpTD algorithms through its unified abstraction.

2. Flexible Buffer Capacity and High Bandwidth: Our architecture tailors the PE group size to the actual tensor density, ρ, ensuring variable buffer capacity. The task transfer mechanism further ensures that tasks benefit from the high bandwidth of local scratchpad memory in each traversed PE.

3. Kernel Fusion Enabled: Our design is free from write conflicts. This is attributed to the compute transfer, which facilitates operations locally within the PE storing the necessary data.

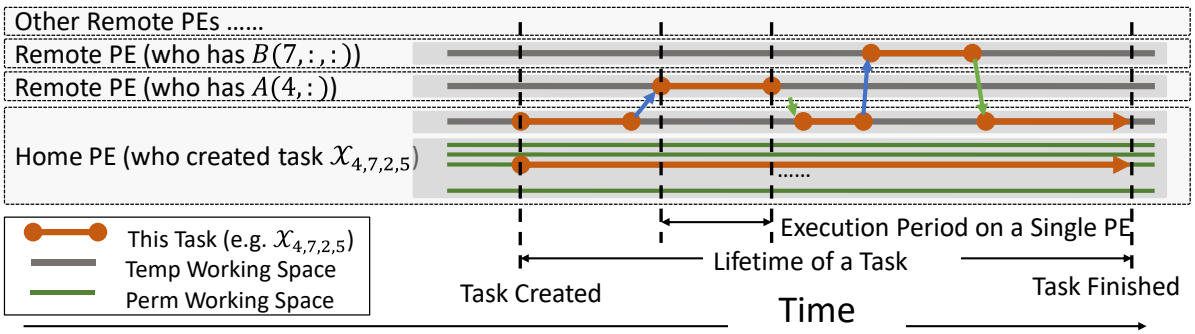


Figure 5.14: The memory trajectory across PEs.

Other Advantages.

1. Latency Insensitivity: All memory accesses during execution are localized, eliminating performance drops due to stalls from non-local memory accesses. If a task requires factor tensor data not present in the current PE, it migrates to another PE possessing the needed data via the NoC (refer to Figure 5.14). This strategy ensures continuous compute unit utilization. Although individual tasks might experience delays in the receive FIFOs, the overall throughput remains unaffected as long as every receive FIFO is non-empty and the compute unit stays active.

2. Lightweight Message: Each message carries a mere 4B of additional overhead (namely, home PE id, slot id, and PC) beyond the effective payload. Additionally, the Perm workspace in the home PE shortens the T_Payload since only data required by the destination PE is transmitted.

5.5 Evaluation

5.5.1 Experimental Setup

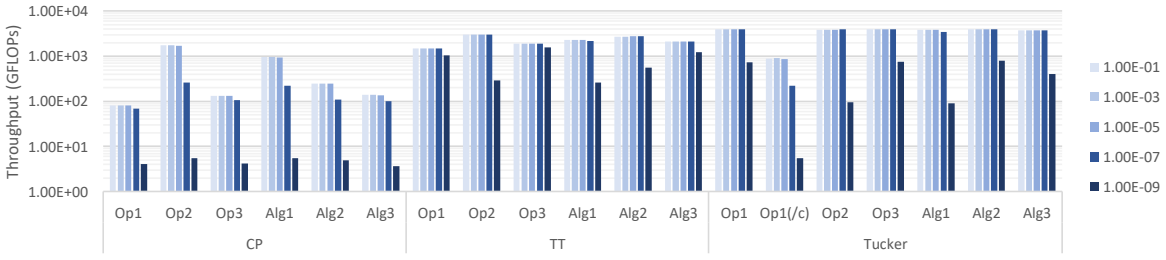


Figure 5.15: STE performance for different algorithms under various input tensor densities. The PE group size is set to 8×8.

System Configuration and Simulation. We evaluate performance through two types of simulations: the single PE level and the PE array level. At the single PE level, we employ send/receive FIFOs and the compute unit, both implemented using Chisel [81]. RTL

simulation allows us to ensure the functional accuracy and wholeness of our instructions and determine their cycle-specific execution times. For a comprehensive performance assessment, we adapt an NoC simulator, specifically BookSim2.0, to model the PE array. New modules are introduced to replicate behaviors of the PE components such as the task injection unit, task receive/send FIFOs, compute unit, and the factor data replacement unit. Within this simulation, the compute unit's actions are abstracted as a countdown timer, with delays set according to the RTL simulation results for a single PE. Traffic in BookSim2.0 encompasses both PE-to-PE and PE-to-memory interactions. To extract detailed data, we incorporate additional performance counters to gauge the compute unit's utilization, the NoC, and the memory controller. It's important to note that we opted against a full-system RTL simulation due to the prohibitively extended simulation time. Instead, we limited RTL simulation to determine the instruction execution times in cycle counts.

For every test bench, our simulator operates across multiple periods to execute a SpTD algorithm, with each period spanning 20K cycles. Simulation concludes once any of the subsequent criteria is achieved: 1) completion of all non-zero entries; 2) the simulator has executed three or more periods and the performance metrics stabilize. Hence, for smaller tensors, all non-zero values are processed. Conversely, for larger tensors, we initially derive steady performance metrics from a subset of sparse entries over restricted simulation durations, which then guides our estimations for the aggregate execution time.

Table 5.2: System configuration.

PE	PE array size: 16×16 Scratchpad Memory: 128 KB per PE Task Send/Receive FIFOs: 16 KB per PE Compute Unit Vector Width: 8 $SlotMax$: 50
NoC	Channel Width: 128 bits
Off-chip Memory	Bandwidth: 24∼96 GB/s

Table 5.3: Area and power breakdown at 22nm and 1GHz.

	Interconnect	Compute	Scratchpad	Memory Controller	Total
Power	26%	29%	42%	3%	62 W
Area	43%	28%	28%	1%	115 mm^2

To estimate the area and power, we employ CACTI [82] for the SRAM simulation, DSENT [83] for the NoC, and McPAT [84] for simulating the DRAM memory controller. The remaining parts of the PE are implemented in RTL and compiled with the Synopsis Design Compiler. For the floating-point unit, we adopt the Berkeley FPU implementation [85]. All these simulations use a 22nm technology library to ensure a fair comparison with contemporary CPU/GPU architectures. Table 5.2 details the system configuration, while Table 5.3 provides a power and area breakdown. It's essential to note that the power values reported represent the peak (worst-case) consumption observed in our tests. Additionally, the default off-chip memory bandwidth is set to 24 GB/s in all experiments, except when juxtaposing performance with CPU/GPU in Section 5.5.3. We chose the lower default bandwidth (24GB/s) as it more aptly demonstrates the efficacy of our architecture. Conversely, the elevated bandwidth (96GB/s) used in the comparison with CPU/GPU ensures equitable evaluation, given the typically high memory bandwidth of CPUs and GPUs.

Table 5.4: Testing cases for each tensor network structure.

Name	Operations or Algorithms	Kernels
$Op1$	$W^T X$	SpLrMM
$Op2$	$(W^T \odot W^T)\Omega$	SpLrMM
$Op3$	$\Omega_X \circ (WH)$	LrSampling
$Alg1$	Masked Square Loss (ALS or Multiplicative Update)	Fused SpLrMMs
$Alg2$	Masked Square Loss (Gradient Descent)	Fused SpLrMM & LrSampling
$Alg3$	KL Divergence (Multiplicative Update)	Fused SpLrMM & LrSampling

Testing Benchmarks. We evaluate a total of 19 cases to showcase our STE's exten-

sive capability in supporting SpTD algorithms. As discussed earlier, the diversity in algorithms is primarily attributed to three factors: i) the network structure; ii) the loss function; iii) the iterative method. In terms of network structure, we focus on the most commonly adopted structures, namely CP, TT, and Tucker. For the latter two factors, we select the ones most widely used in the industry (refer to Table 5.4). It's important to note that for the Tucker structure, an additional variant $Op1(/c)$ exists for $Op1$. This variant represents a modified SpLrMM kernel, with its core factor tensor omitted. This version essentially equates to SpTTMc.

Unless mentioned otherwise, we utilize randomly generated sparse tensor data in our simulations. This choice is motivated by our aim to explore the sparsity spectrum systematically, thus offering a more comprehensive understanding. Additionally, performance evaluations on CPU/GPU are carried out using actual real-world tensor datasets. For the sake of simplicity, we have fixed the rank values at 16 throughout this section.

Table 5.5: Bounding factors and solution suggestions.

	Bounding Factor	Solution Suggestions
(Comp.)	Compute unit	Increase the compute resources
(NoC)	NoC	Decrease PE group size, use compute transfer, or increase NoC channel width
(Mem.1)	Load sparse entries (i.e. $(\boldsymbol{\mathcal{X}}_{ijk}, i, j, k)$)	Increase off-chip memory bandwidth
(Mem.2)	Load factor tensor data (e.g. $A(i,:)$,$B(j,:,:)$)	Increase PE group size, or increase on-chip memory capacity

5.5.2 Overall Performance

Figure 5.15 depicts the overall performance in FLOPs across various SpTD algorithms and input data densities, ranging from 10^{-1} to 10^{-9}. The system's performance spans from 82 GFLOPs to 3.9 TFLOPs, covering a diverse set of algorithms and sparsity levels. For context, the theoretical peak performance of STE is 4 TFLOPs. It's evident that

algorithms based on the CP structure typically yield a lower throughput. This outcome is attributed to the fact that CP-based algorithms usually encompass fewer computations for each sparse entry as compared to their TT and Tucker counterparts. Consequently, the performance of CP-based algorithms is more constrained by the process of fetching sparse entries from off-chip memory, unlike the computation-intensive nature observed in TT- and Tucker-based algorithms.

In our analysis, we identify four primary factors that potentially restrict system performance, as detailed in Table 5.5. The performance bottleneck can emerge from the compute unit (Comp.), NoC, or the process of loading either sparse entries (Mem.1) or factor tensor data (Mem.2) from off-chip memory. The dominant factor affecting performance is contingent upon the chosen algorithm, input tensor sparsity, and specific system configurations, including the PE group size and off-chip memory bandwidth.

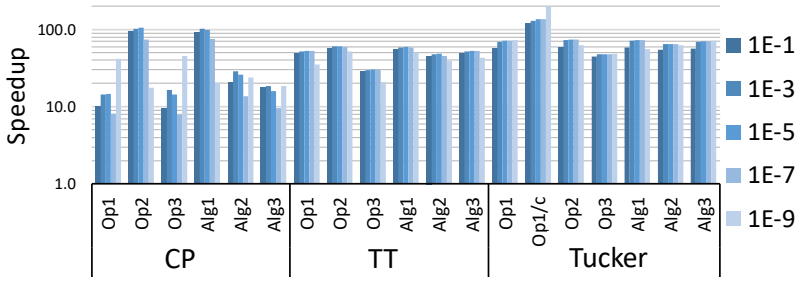


Figure 5.16: Speedup over CPU.

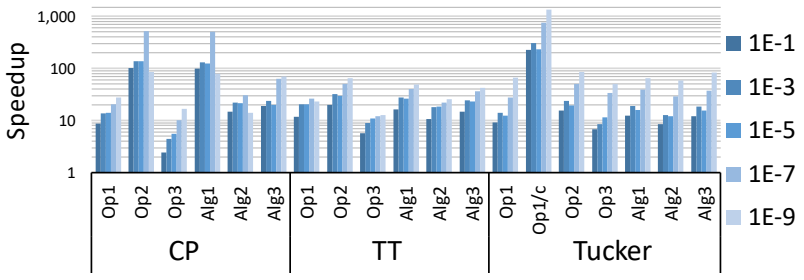


Figure 5.17: Speedup over GPU.

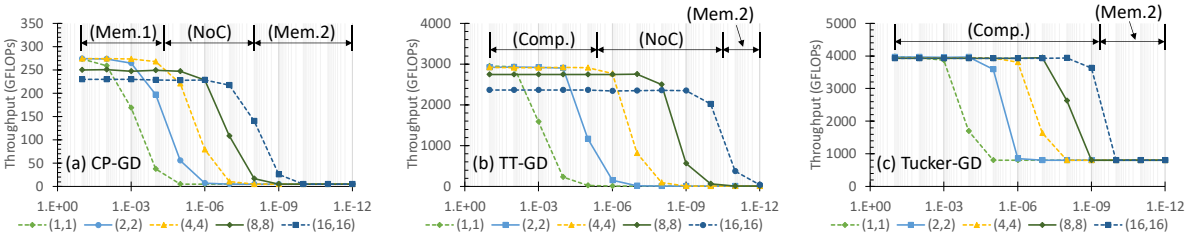


Figure 5.18: Performance curve with increasing input tensor sparsity and PE group size.

5.5.3 Comparison with CPU and GPU

To benchmark STE against CPU and GPU implementations, we employ Taco [86] to generate codes corresponding to $Op1$, $Op2$, and $Op3$. Since Taco lacks support for kernel fusion, the cases $Alg1$, $Alg2$, and $Alg3$ are implemented by invoking $Op1$, $Op2$, and $Op3$ separately. While there exist specialized high-performance routines for SpTD on both CPU (like SPLATT [71]) and GPU (such as B. Liu et al. [72]), these only cater to one or two specific kernels, thus limiting the scope for a comprehensive comparison. However, it's worth noting that Taco-generated codes exhibit performance levels comparable to hand-tuned kernels [86]. The CPU we benchmarked with is 2× Intel(R) Xeon(R) CPU E5-2620 v4, paired with four DDR4 2133 MHz memory modules, and we deployed 32 threads for execution. Each CPU comprises a cache size of 22.25MB (accumulative of L1, L2, and L3), culminating in a system-wide cache of 44.5MB with the dual CPUs. Our chosen GPU for benchmarking is the NVIDIA Titan V. It should be noted that while the default off-chip memory bandwidth of STE stands at 24 GB/s, the bandwidths for the CPU and GPU can escalate to 96 GB/s and over 500 GB/s, respectively. To level the playing field in our comparison, we've enhanced STE's off-chip memory bandwidth to 96 GB/s for this section. The CPU and GPU platforms peak at 600 GFLOPs and 15.7 TFLOPs, respectively.

Figures 5.16 and 5.17 illustrate the speedup of STE over the CPU and GPU using

synthetic data, respectively. It is essential to note that this is a juxtaposition of simulated results from our proposed architecture against real-world empirical results from CPUs and GPUs. Although the simulation may exhibit minor deviations from the actual performance due to inherent simplifications, it distinctly underscores the efficacy of our architectural design. On average, STE realizes a speedup of 45× over the CPU and 29× over the GPU. Specifically, remarkable acceleration is observed in CP-$Op2$, CP-$Alg1$, and Tucker-$Op1$(/c) scenarios, primarily because the performance in these cases is majorly influenced by the "Mem.2" factor, underscoring STE's distinct advantage.

Moreover, Table 5.6 presents the speedup of STE over CPU/GPU using real-world datasets, mirroring the results observed with synthetic data. The data origins are as follows: DNN represents a reshaped weight matrix from a pruned fully-connected layer in deep neural networks; Nell2 [87] is sourced from a natural language dataset; NIPS [88] is a tensor capturing publication statistics from the Annual Conference on Neural Information Processing Systems; and Email [79] is a tensor logging email transmission statistics. This table further showcases the broader implications of our work across various domains. The DNN-CP-Op1 scenario, bound by memory, warrants further elucidation: its relative speedup over the CPU/GPU is slightly reduced compared to synthetic data. This can be attributed to the tensor in this scenario being both denser and of smaller size, which enhances the CPU/GPU performance by optimizing cache utilization. Conversely, for DNN-Tucker-Op1, which is computation-bound, STE continues to maintain a substantial performance edge.

5.5.4 Analysis of Tensor Sparsity and PE Group Size

For this analysis, we focus solely on $Alg2$ (GD) to streamline our experimental scope. As illustrated in Figure 5.18, there's a discernible decrease in performance following a

Table 5.6: Speedup over CPU and GPU using real-world data.

Name	Alg	Size	Density	STE/CPU	STE/GPU
DNN	CP-Op1	512×512×392	3.0E-01	3.3×	3.4×
Nell2	CP-Op1	12K×9K×28K	2.4E-05	11.1×	14.2×
DNN	Tucker-Op1	512×512×392	3.0E-01	50.1×	7.0×
Nell2	Tucker-Op1	12K×9K×28K	2.4E-05	70.6×	14.4×
NIPS	TT-Op1	2K×3K×14K×17	1.8E-06	40.2×	24.1×
Email	TT-Op1	6K×6K×224K×1K	5.5E-09	33.2×	20.0×

"ladder-shaped" trajectory as tensor sparsity intensifies. This pattern mirrors observations from other algorithms as seen in Figure 5.15. These "ladder-shaped" curves can be segmented into three distinct regions, possibly resulting from varying bounding factors in tandem with the increase in sparsity.

To explore this supposition further, we assessed the resource allocation across various hardware components, as illustrated in Figure 5.19. These findings have been annotated in Figure 5.18 for clarity. A conspicuous observation is that in scenarios of extreme sparsity, the performance is often limited by the factor tensor loading (Mem.2). This limitation arises due to an increased demand for buffer capacity that outstrips the PE group's thresholds. Larger PE group sizes proffer augmented buffer capacity, which in turn postpones the initiation of the performance decline. However, this advantage does come at a cost. An increase in PE group size inherently augments the inter-PE communication distance, thereby negatively impacting performance when the NoC becomes the binding constraint, as reflected in the regions demarcated with NoC in Figure 5.18. Therefore, there's no one-size-fits-all solution when configuring the PE group size, given the variable sparsity levels. A pragmatic approach would entail tailoring the PE group size to the actual sparsity level of the input tensors.

We glean three noteworthy observations from Figure 5.18.

1. At lower sparsity levels, CP-GD's performance is constrained by Mem.1 (loading

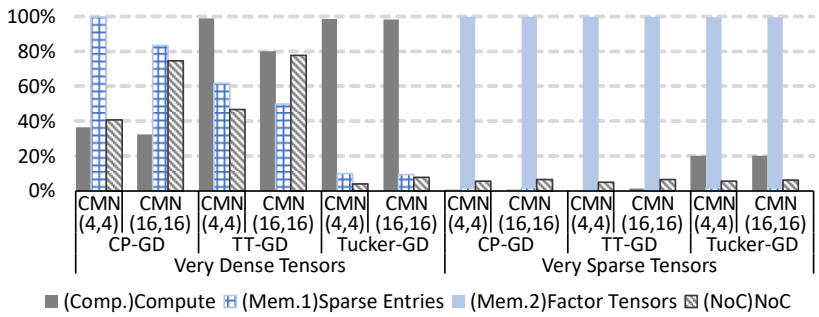


Figure 5.19: Resource utilization breakdown.

Table 5.7: Computation and memory costs for each sparse entry. Mem.2 can be amortized due to the inter-sparse entry reuse. MACs: multiplication-and-accumulation operations.

Algorithm	Comp. (MACs)	NoC (Bytes)	Mem.1 (Bytes)	Mem.2 (Bytes)
CP-GD	161	768	14	768
TT-GD	3169	1792	16	8704
Tucker-GD	26225	768	14	66304

sparse entries). In contrast, TT-GD and Tucker-GD are bounded by Comp. This can be attributed to the higher computational requirements of TT-GD and Tucker-GD for each sparse entry, as delineated in Table 5.7.

2. Surprisingly, Tucker-GD does not exhibit a NoC bound region. A possible explanation lies in its significantly elevated MACs/traffic ratio, as further elucidated in Table 5.7.

3. Tucker-GD's throughput remains non-zero, even in the face of heightened sparsity. Revisiting Figure 5.2, we can attribute this resilience to the core tensor inherent in the Tucker decomposition. Its applicability across all sparse entries ensures a baseline computational consistency.

5.5.5 Analysis of Memory Optimization

To discern the implications of varying off-chip memory bandwidths on performance, we scaled the bandwidth from the baseline 24 GB/s to 2×, 3×, and 4× its initial value. The resulting performance variations are depicted in Figure 5.20. A deeper understanding of these fluctuations necessitates an intersectional reading of Table 5.5 and Figure 5.18.

In dense tensor scenarios like CP-GD ($\rho = 10^{-1}$ and $\rho = 10^{-5}$), we observe an initial uptick in performance which plateaus at the 2× bandwidth. This surge is a direct consequence of Mem.1 restrictions at the onset, with the ensuing saturation arising from the subsequent NoC bounds. On the other hand, for exceedingly sparse tensors such as CP-GD with $\rho = 10^{-9}$, our testing revealed a consistent performance uptrend, given the perpetual bounding by Mem.2. Contrarily, TT-GD and Tucker-GD showed no perceptible performance shifts for densities $\rho = 10^{-1}$ and $\rho = 10^{-5}$, underscoring their computation-intensive nature over memory dependencies.

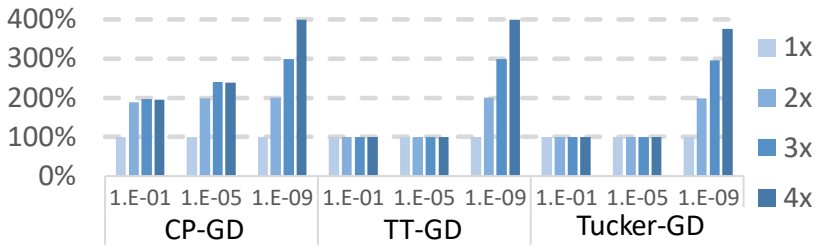


Figure 5.20: Normalized performance with increased off-chip memory bandwidth. The PE group size is set to 4×4.

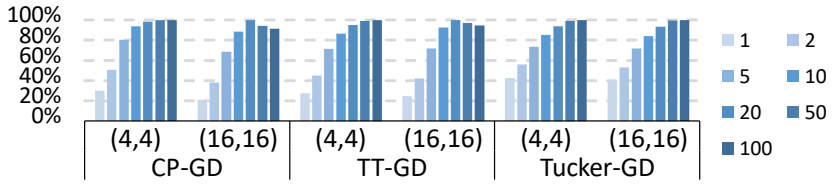


Figure 5.21: Normalized performance with increased $SlotMax$. The tensor density is 10^{-1}.

The influence of $SlotMax$ is illustrated in Figure 5.21. A smaller $SlotMax$ trans-

lates to fewer on-the-fly tasks, rendering the compute unit more susceptible to idleness. Notably, a conspicuous performance penalty surfaces when $SlotMax < 20$. As $SlotMax$ surpasses 50, any further performance enhancements cease. Hence, we opt for $SlotMax = 50$ in our architecture design.

5.5.6 Analysis of Compute Optimization

We ascertain the impact of kernel fusion and compute transfer optimization in Figure 5.22. In absence of kernel fusion optimization, kernels must execute individually, engendering redundant computations and off-chip accesses. It's essential to emphasize that $Op1$, $Op2$, and $Op3$ are impervious to the advantages of kernel fusion due to their solitary nature. Without compute transfer optimization, computation is confined solely to the home PE, relegating remote PEs to respond solely to rudimentary read/write requests (emulated through tasks) via the NoC. This approach generates an on-chip traffic volume akin to conventional reads/writes from/to the shared global buffer on conventional architectures, making it a viable approximation. The resultant performance nosedive is notably profound, occasionally plummeting to a mere 8

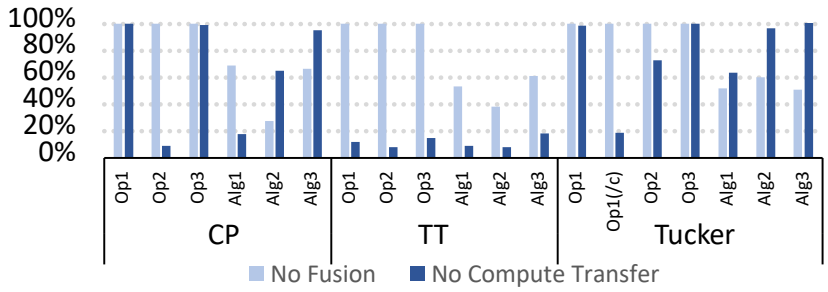


Figure 5.22: Normalized performance influenced by kernel fusion or compute transfer. The tensor density and PE group size are set to 10^{-5} and 8×8, respectively.

5.6 Conclusion

We identify that the challenges in processing sparse tensor decomposition (SpTD) include the algorithm diversity, requirements for buffer flexibility and data bandwidth due to variable sparsity, and the hardness to fuse kernels. To address these issues, we propose a unified abstraction, namely two general sparse kernels (i.e., SpLrMM and LrSampling) and a unified execution framework that can accommodate most algorithms with kernel fusion. Then, we design a specialized accelerator, STE, to implement our top-down solution. The PE-interactive architecture enables the sharing of local memory capacity/bandwidth of each PE and avoids write conflicts during kernel fusion. The deadlock is also eliminated by identifying and satisfying three requirements during task mapping. Through extensive experiments, we demonstrate an average speedup of 45× over CPU and 29× over GPU. Comprehensive analysis of the impact of tensor sparsity, PE group size, and memory/compute optimizations are further presented to give design guidance. Our design can stimulate more researches in designing specialized architectures for high-performance tensor decomposition.

Chapter 6

Summary

(Key, Value) pairs are one of the most prominent forms of data structures in modern computer systems, and the operations on them represent a significant portion of computation in many domains, including AI, databases, genomic processing, graph analysis, and scientific computing. For each of these domains, a lot of research on architecture design and hardware acceleration can be found. However, these research efforts are rarely adopted in the real world due to the high cost of developing new DSAs or processors. Economic viability prevents their wider adoption except in a few areas with huge market demands, such as AI. In this thesis, we aim to gain a more essential understanding and abstraction of (key, value) pair operations. Subsequently, we strive to develop a more general and reusable design that can be applied across multiple domains.

We first address the problem of handling operations on a data structure of ordered (key, value) pairs. Our journey begins with a simple but fundamental operation: the merge-style operation that combines two sorted arrays into one, allowing for customizable behavior when dealing with matched and mismatched tuples. This operation encompasses several widely recognized tasks, such as sparse additions and multiplications, database joins, and sorting. Implementing such operations in hardware posed significant

challenges, each of which we tackled. Firstly, we introduced a novel abstraction known as the restricted zip machine (RZM), capable of representing all merge-style operations. Secondly, we presented four single instruction, multiple data (SIMD) primitives that can be employed to implement any RZM operation. Thirdly, we devised a hardware design for these four SIMD primitives with a complexity of $O(V \log(V))$, where V represents the SIMD width. In contrast to existing work, this design covers a broader range of applications while reducing total hardware costs from $O(V^2)$ to $O(V \log(V))$. The resulting design is adaptable for use with CPUs, GPUs, or domain-specific accelerators. Upon integration into CPUs as SIMD primitives (as demonstrated in our evaluation), we achieved significant speedup ranging from 4× to 8× across various representative kernels. These include set operations (8.4×), database joins (7.3×), sparse vector/matrix/tensor addition/multiplication involving real/complex numbers (6.5×), merge sort (8.0× compared to scalar, 3.4× compared to state-of-the-art SIMD), and SpGEMM (4.4× improvement over the best result in the baseline collection).

Our second step involves addressing higher-level ordered (Key, Value) data structures and their operations beyond mere array merging. Interestingly, the addition of new hardware isn't necessary. The accelerated merge-style operation (which we already possess) emerges as a potent tool, enabling significant enhancements to higher-level operations through algorithm redesign alone. We present several examples:

1. Heap: We reimagine the conventional binary heap as a combination of large and small heaps. This restructuring allows us to harness the power of the merge-style operation SIMD primitives, resulting in a theoretically achievable $O(V)$ speedup, where V denotes the SIMD width. In practice, on a CPU with $V = 16$, we attain speed-ups of around 7× for batch operations and approximately 3× for single-element operations.

2. Batched Binary Search: Utilizing a combination of SORT-pattern and Interpolate-pattern merge-style operations, batched binary search demonstrates speedups ranging from 1.6× to 2× when the data is located in L2 cache or off-chip memory.

3. K-way Additive Merge: We demonstrate that a meticulously designed merge-order scheduling algorithm can proficiently execute K-way additive merge operations. This design guarantees a per-sparse entry time complexity, bounded by the mature information of the sparse pattern.

For unordered operations on (Key, Value) pairs, one of the most commonly used is the reduce-by-key operation. However, this operation is known to be unfriendly to hardware due to issues with parallelism, memory access patterns/traffic, and space efficiency. We recognize that the key problem lies in deciding whether to accept or reject duplications. This decision leads to dramatic distinctions in parallelism, memory access, and space characteristics. We introduce a new hardware-friendly algorithm that capitalizes on the uniformity of a data organization called the "sorted, deduplicated hash array (SDHA)." This data organization smoothly transitions between two statuses and combines the best aspects of both. Importantly, this approach is not only empirical but also has provably near-optimal space and traffic characteristics. Additionally, we present a hardware design that efficiently supports the operations required by this new algorithm. Evaluation demonstrates that our design consistently achieves superior performance compared to existing methods across a wide range of input distributions.

Finally, we delve into a data analysis method on (Key, Value) pairs for multi-field keys named "sparse tensor decomposition (SpDT)," where the sole assumption is that the relation between the key and value can be perceived as samples from a low-rank tensor. We propose a hardware design for SpDT that fulfills its data reuse requirement by enabling private buffers for each processing element (PE) to be logically shared through

light-weight task migration between PEs. This design ultimately reduces traffic over the Network on Chip (NoC) and off-chip memory, thereby enhancing performance.

With all the work mentioned above, we have presented a comprehensive study on how to define expressive abstractions and efficient hardware designs for (Key, Value) pair operations. These operations, in turn, constitute a substantial portion of computations across various domains. We firmly believe that our efforts will serve as a catalyst for further research in this field, and the designs introduced throughout this series of work will become indispensable components of future general-purpose processors and domain-specific accelerators.

Lessons and Future Work

There are two types of explorations in designing specialized hardware: one is a standalone accelerator that completes the entire workload without assistance from external resources, and the other is to extend composable low-level primitives that can be easily integrated with other existing primitives in general-purpose processors. In most cases, the latter style is commercially more promising, as investments in hardware design tend to be reusable in a wider range of applications. Our series of work mostly falls into the second category, and we have witnessed how this choice pays off: by simply rewriting a few tens or hundreds of lines of software code, the merge-style operation primitives designed bring about up to an 8x speedup, spanning from scientific workloads like sparse computing and graph analysis to business workloads like database operations and "daily utilities" such as sorting, searching, and priority queues. This result also highlights an important fact:

Future direction: The primitives (especially SIMD primitives) provided by current mainstream ISAs are still very incomplete, even for very common tasks with vast potential users. There is still plenty of room for improvement in this direction.

Now, let's revisit the (key, value) operations. I believe there is one remaining aspect we have yet to explore: (key, value) operations with complex data dependencies driven by the keys. These operations include one of the most crucial kernels in scientific computing and engineering design: sparse Gaussian elimination, a key component for solving linear equations. None of the methods proposed in this thesis, nor the existing massively parallel architectures like GPUs, can handle this problem flawlessly. For instance, the GPU library cuSolver sometimes re-offloads tasks back to CPUs. The reason behind this challenge is that the "data dependency depends on data" aspect is too intricate to be expressed in a SIMD fashion. Therefore, it would be particularly intriguing to devise a method for designing hardware capable of handling this challenging-to-exploit parallelism.

Once again, we emphasize our preference for "reusable low-level primitives" over "standalone accelerators" in this scenario. Thus, what we are ultimately seeking is a way to abstract the capability for managing dependencies (rather than computation) into a "primitive."

Future Direction: How can we design, abstract, and ultimately implement the capability to manage complex parallel dependencies as a form of reusable low-level primitive, similar to what we have done for computation primitives?

Bibliography

[1] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, Extensor, in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, (New York, NY, USA), pp. 319–333, ACM, 10122019.

[2] Flexminer: A pattern-aware accelerator for graph pattern mining, .

[3] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, Towards general purpose acceleration by exploiting common data-dependence forms, in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, (New York, NY, USA), pp. 924–939, ACM, 10122019.

[4] R. Hojabr, A. Sedaghati, A. Sharifian, A. Khonsari, and A. Shriraman, Spaghetti: Streaming accelerators for highly sparse gemm on fpgas, in 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 84–96, IEEE, 2/27/2021 - 3/3/2021.

[5] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product, in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 766–780, IEEE, 10/17/2020 - 10/21/2020.

[6] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, Gamma: leveraging gustavson's algorithm to accelerate sparse matrix multiplication, in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (T. Sherwood, E. Berger, and C. Kozyrakis, eds.), (New York, NY, USA), pp. 687–701, ACM, 04192021.

[7] J. Casper and K. Olukotun, Hardware acceleration of database operations, in FPGA '14, (New York), ACM, 2014.

[8] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, and S. W. Keckler and Z. Zhang, C24-4 snap: A 1.67 – 21.55tops/w sparse neural acceleration processor for unstructured sparse deep neural network inference in 16nm cmos, .

[9] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication."

[10] Benjamin Schlegel,Thomas Willhalm, Wolfgang Lehner, Fast sorted-set intersection using simd instructions, .

[11] S. Han, L. Zou, and J. X. Yu, Speeding up set intersections in graph algorithms using simd instructions, in Proceedings of the 2018 International Conference on Management of Data (G. Das, C. Jermaine, and P. Bernstein, eds.), (New York, NY, USA), pp. 1587–1602, ACM, 05272018.

[12] H. Inoue, Faster set intersection with simd instructions by reducing branch mispredictions, .

[13] K. E. Batcher, Sorting networks and their applications, in Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring), (New York, New York, USA), ACM Press, 1968.

[14] "Intel intrinsics guide." `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

[15] H. Inoue and K. Taura, Simd- and cache-friendly algorithm for sorting an array of structures, Proceedings of the VLDB Endowment 8 (2015), no. 11 1274–1285.

[16] B. Bramas, A fast vectorized sorting implementation based on the arm scalable vector extension (sve), ArXiv abs/2105.07782 (2021).

[17] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, Sort vs. hash revisited, Proceedings of the VLDB Endowment 2 (2009), no. 2 1378–1389.

[18] "Eigen is a c++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.." `https://eigen.tuxfamily.org/index.php?title=Main_Page`.

[19] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, High-performance sparse matrix-matrix products on intel knl and multicore architectures, in Proceedings of the 47th International Conference on Parallel Processing Companion, (New York, NY, USA), pp. 1–10, ACM, 08132018.

[20] S. Buss and A. Knop, Strategies for Stable Merge Sorting, pp. 1272–1290. https://epubs.siam.org/doi/pdf/10.1137/1.9781611975482.78.

[21] A. Mukkara, N. Beckmann, and D. Sanchez, Phi: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates, in Proceedings of the 52nd Annual IEEE/ACM International Symposium on

Microarchitecture, MICRO '52, (New York, NY, USA), p. 1009–1022, Association for Computing Machinery, 2019.

[22] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, Graphicionado: A high-performance and energy-efficient accelerator for graph analytics, in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1–13, 2016.

[23] H. Vandierendonck, Graptor: Efficient pull and push style vectorized graph processing, in Proceedings of the 34th ACM International Conference on Supercomputing, ICS '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[24] G. Marçais and C. Kingsford, A fast, lock-free approach for efficient parallel counting of occurrences of k-mers, Bioinformatics 27 (2011), no. 6 764–770.

[25] D. Mapleson, G. Garcia Accinelli, G. Kettleborough, J. Wright, and B. J. Clavijo, Kat: a k-mer analysis toolkit to quality control ngs datasets and genome assemblies, Bioinformatics 33 (2017), no. 4 574–576.

[26] P. Melsted and J. K. Pritchard, Efficient counting of k-mers in dna sequences using a bloom filter, BMC bioinformatics 12 (2011), no. 1 1–7.

[27] L. Kaplinski, M. Lepamets, and M. Remm, Genometester4: a toolkit for performing basic set operations-union, intersection and complement on k-mer lists, Gigascience 4 (2015), no. 1 s13742–015.

[28] R. S. Roy, D. Bhattacharya, and A. Schliep, Turtle: Identifying frequent k-mers with cache-efficient algorithms, Bioinformatics 30 (2014), no. 14 1950–1957.

[29] M. Kokot, M. Dlugosz, and S. Deorowicz, Kmc 3: counting and manipulating k-mer statistics, Bioinformatics 33 (2017) 2759–2761.

[30] N. Bell, S. Dalton, and L. N. Olson, Exposing fine-grained parallelism in algebraic multigrid methods, SIAM Journal on Scientific Computing 34 (2012), no. 4 C123–C152.

[31] S. Dalton, L. Olson, and N. Bell, Optimizing sparse matrix—matrix multiplication for the gpu, ACM Trans. Math. Softw. 41 (oct, 2015).

[32] F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling, and U. Naumann, Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging, SIAM Journal on Scientific Computing 37 (01, 2015) C54–C71.

[33] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, Adaptive sparse matrix-matrix multiplication on the gpu, in Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPoPP '19, (New York, NY, USA), p. 68–81, Association for Computing Machinery, 2019.

[34] W. Liu and B. Vinter, An efficient gpu general sparse matrix-matrix multiplication for irregular data, in 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 370–381, 2014.

[35] Y. Nagasaka, A. Nukada, and S. Matsuoka, High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu, in 2017 46th International Conference on Parallel Processing (ICPP), pp. 101–110, 2017.

[36] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, High-performance sparse matrix-matrix products on intel knl and multicore architectures, in Proceedings of the 47th International Conference on Parallel Processing Companion, ICPP '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[37] M. Deveci, C. Trott, and S. Rajamanickam, Multi-threaded sparse matrix-matrix multiplication for many-core and gpu architectures, Parallel Computing 78 (01, 2018).

[38] M. Deveci, C. Trott, and S. Rajamanickam, Performance-portable sparse matrix-matrix multiplication for many-core architectures, in 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 693–702, 2017.

[39] P. Memarzia, S. Ray, and V. Bhavsar, A six-dimensional analysis of in-memory aggregation, 03, 2019.

[40] O. Polychroniou and K. A. Ross, A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort, in Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, (New York, NY, USA), p. 755–766, Association for Computing Machinery, 2014.

[41] T. Karnagel, R. Müller, and G. M. Lohman, Optimizing gpu-accelerated group-by and aggregation, in ADMS@VLDB, 2015.

[42] P. Jiang and G. Agrawal, Efficient simd and mimd parallelization of hash-based aggregation by conflict mitigation, in Proceedings of the International Conference on Supercomputing, (New York, NY, USA), ACM, 2017.

[43] I. Absalyamov, P. Budhkar, S. Windh, R. J. Halstead, W. A. Najjar, and V. J. Tsotras, Fpga-accelerated group-by aggregation using synchronizing caches, in

Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[44] D. G. Tomé, T. Gubner, M. Raasveldt, E. Rozenberg, and P. A. Boncz, Optimizing group-by and aggregation using gpu-cpu co-processing, in ADMS@VLDB, 2018.

[45] I. Müller, A. Lacurie, P. Sanders, W. Lehner, and F. Färber, Cache-efficient aggregation: Hashing is sorting, 06, 2015.

[46] N. E. Young, Online Paging and Caching, pp. 1457–1461. Springer New York, New York, NY, 2016.

[47] D. D. Sleator and R. E. Tarjan, Amortized efficiency of list update and paging rules, Commun. ACM 28 (feb, 1985) 202–208.

[48] S.-W. Jun, S. Xu, and Arvind, Terabyte sort on fpga-accelerated flash storage, in 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 17–24, 2017.

[49] 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, pp. 214–219, 2014.

[50] Y. Yang, S. R. Kuppannagari, and V. K. Prasanna, A high throughput parallel hash table accelerator on hbm-enabled fpgas, in 2020 International Conference on Field-Programmable Technology (ICFPT), pp. 148–153, 2020.

[51] B. Romanous, S. Windh, I. Absalyamov, P. Budhkar, R. Halstead, W. Najjar, and V. Tsotras, Efficient local locking for massively multithreaded in-memory hash-based operators, The VLDB Journal 30 (feb, 2021) 333–359.

[52] B. W. Bader, T. G. Kolda, and R. A. Harshman, Temporal analysis of social networks using three-way dedicom., tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); Sandia ..., 2006.

[53] S. Sizov, S. Staab, and T. Franz, Analysis of Social Networks by Tensor Decomposition, pp. 45–58. Springer US, Boston, MA, 2010.

[54] E. Acar, S. A. Çamtepe, M. S. Krishnamoorthy, and B. Yener, Modeling and multiway analysis of chatroom tensors, in Intelligence and Security Informatics (P. Kantor, G. Muresan, F. Roberts, D. D. Zeng, F.-Y. Wang, H. Chen, and R. C. Merkle, eds.), (Berlin, Heidelberg), pp. 256–268, Springer Berlin Heidelberg, 2005.

[55] M. M. Baskaran, T. Henretty, J. Ezick, R. Lethin, and D. Bruns-Smith, Enhancing network visibility and security through tensor analysis, Future Generations Computer Systems (2, 2019).

[56] K. Xie, L. Wang, X. Wang, G. Xie, J. Wen, and G. Zhang, Accurate recovery of internet traffic data: A tensor completion approach, in IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications, pp. 1–9, April, 2016.

[57] I. Perros, R. Chen, R. Vuduc, and J. Sun, Sparse hierarchical tucker factorization and its application to healthcare, in , pp. 943–948, 11, 2015.

[58] J. Ho, J. Ghosh, and J. Sun, Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization, Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (08, 2014).

[59] J. He, Q. Liu, A. Christodoulou, C. Ma, F. Lam, and Z.-P. Liang, Accelerated high-dimensional mr imaging with sparse sampling using low-rank tensors, IEEE Transactions on Medical Imaging 35 (04, 2016) 2119–2129.

[60] S. Gandy, B. Recht, and I. Yamada, Tensor completion and low-n-rank tensor recovery via convex optimization, Inverse Problems 27 (jan, 2011) 025010.

[61] J. Liu, P. Musialski, P. Wonka, and J. Ye, Tensor completion for estimating missing values in visual data, IEEE Transactions on Pattern Analysis and Machine Intelligence 35 (Jan, 2013) 208–220.

[62] S. Smith, K. Huang, N. D. Sidiropoulos, and G. Karypis, Streaming Tensor Factorization for Infinite Data Sources, pp. 81–89. , . https://epubs.siam.org/doi/pdf/10.1137/1.9781611975321.10.

[63] J. Sun, D. Tao, and C. Faloutsos, Beyond streams and graphs: Dynamic tensor analysis, in Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06, (New York, NY, USA), pp. 374–383, ACM, 2006.

[64] R. Orus, A practical introduction to tensor networks: Matrix product states and projected entangled pair states, Annals of Physics 349 (06, 2013).

[65] D. Hong, T. G. Kolda, and J. A. Duersch, Generalized canonical polyadic tensor decomposition, SIAM Review 62 (2020), no. 1 133–163, [arXiv:1808.0745].

[66] N. Park, B. Jeon, J. Lee, and U. Kang, Bigtensor: Mining billion-scale tensor made easy, in Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, CIKM '16, (New York, NY, USA), pp. 2457–2460, ACM, 2016.

[67] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, Gigatensor: Scaling tensor analysis up by 100 times - algorithms and discoveries, in Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, (New York, NY, USA), pp. 316–324, ACM, 2012.

[68] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, Haten2: Billion-scale tensor decompositions, in 2015 IEEE 31st International Conference on Data Engineering, pp. 1047–1058, April, 2015.

[69] O. Kaya and B. Uçar, Scalable sparse tensor decompositions in distributed memory systems, in SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11, Nov, 2015.

[70] S. Smith and G. Karypis, A medium-grained algorithm for sparse tensor factorization, in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 902–911, May, 2016.

[71] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, Splatt: Efficient and parallel sparse tensor-matrix multiplication, in 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 61–70, May, 2015.

[72] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, A unified optimization approach for sparse tensor operations on gpus, in 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 47–57, Sep., 2017.

[73] J. Li, Y. Ma, and R. Vuduc, ParTI! : A parallel tensor infrastructure for multicore cpus and gpus, Oct, 2018.

[74] J. H. Choi and S. V. N. Vishwanathan, Dfacto: Distributed factorization of tensors, in Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1, NIPS'14, (Cambridge, MA, USA), pp. 1296–1304, MIT Press, 2014.

[75] S. Smith and G. Karypis, Accelerating the tucker decomposition with compressed sparse tensors, in Euro-Par 2017: Parallel Processing (F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, eds.), (Cham), pp. 653–668, Springer International Publishing, 2017.

[76] O. Kaya and B. Uçar, High performance parallel algorithms for the tucker decomposition of sparse tensors, in 2016 45th International Conference on Parallel Processing (ICPP), pp. 103–112, Aug, 2016.

[77] A. Beutel, P. P. Talukdar, A. Kumar, C. Faloutsos, E. E. Papalexakis, and E. P. Xing, Flexifact: Scalable flexible factorization of coupled tensors on hadoop, in Proceedings of the 2014 SIAM International Conference on Data Mining, pp. 109–117, SIAM, 2014.

[78] S. Smith, J. Park, and G. Karypis, An exploration of optimization algorithms for high performance tensor completion, in SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 359–371, Nov, 2016.

[79] J. Shetty and J. Adibi, The enron email dataset database schema and brief statistical report, Information sciences institute technical report, University of Southern California 4 (2004).

[80] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations, 04, 2020.

[81] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, Chisel: Constructing hardware in a scala embedded language, in DAC Design Automation Conference 2012, pp. 1212–1221, June, 2012.

[82] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, Cacti 6.0: A tool to model large caches, HP Laboratories (01, 2009).

[83] C. Sun, C. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. Peh, and V. Stojanovic, Dsent - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling, in 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip, pp. 201–210, May, 2012.

[84] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures, in MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 469–480, 2009.

[85] "Berkeley fpu." `https://github.com/ucb-bar/berkeley-hardfloat`. Accessed: 2019-09-30.

[86] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, Taco: A tool to generate tensor algebra kernels, in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 943–948, Oct, 2017.

[87] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell, Toward an architecture for never-ending language learning., in AAAI, vol. 5, p. 3, 2010.

[88] A. Globerson, G. Chechik, F. Pereira, and N. Tishby, Euclidean Embedding of Co-occurrence Data, The Journal of Machine Learning Research 8 (2007) 2265–2295.