A One-for-All and $O(V \log(V))$-Cost Solution for Parallel Merge Style Operations on Sorted Key-Value Arrays

Bangyan Wang
University of California, Santa Barbara
USA
wangbangyan@gmail.com

Lei Deng
Tsinghua University
China
leideng@mail.tsinghua.edu.cn

Fei Sun
Alibaba DAMO Academy
China
f.sun@alibaba-inc.com

Guohao Dai
Tsinghua University
China
daiguohao@mail.tsinghua.edu.cn

Liu Liu
University of California, Santa Barbara
USA
liu|iu@ucsb.edu

Yu Wang
Tsinghua University
China
yu-wang@tsinghua.edu.cn

Yuan Xie
University of California, Santa Barbara
USA
yuanxie@ucsb.edu

ABSTRACT

The processing of sorted key-value arrays using a "merge style operation (MSO)" is a very basic and important problem in domains like scientific computing, deep learning, database, graph analysis, sorting, set-operation etc. MSOs dominate the execution time in some important applications like SpGEMM and graph mining. For example, sparse vector addition as an MSO takes up to 98% execution time in SpGEMM in our experiment. For this reason, accelerating MSOs on CPU, GPU, and accelerators using parallel execution has been extensively studied but the solutions in prior work have three major limitations. (1) They treat different MSOs as isolated problems using incompatible methods and an unified solution is still lacking. (2) They do not have the flexibility to support variable key/value sizes and value calculations in the runtime given a fixed hardware design. (3) They require a quadratic hardware cost ($O(V^2)$) for given parallelism V in most cases.

To address above three limitations, we make the following efforts. (1) We present a one-for-all solution to support all interested MSOs based on a unified abstraction model "restricted zip machine (RZM)". (2) We propose a set of composable and parallel primitives for RZM to provide the flexibility to support variable key/value sizes and value calculations. (3) We provide the hardware design to implement the proposed primitives using only $O(V \log(V))$ resource. With the above techniques, a flexible and efficient solution for MSOs has been built. Our design can be used either as a drop-in replacement of the merge unit in prior accelerators to reduce the cost from $O(V^2)$ to $O(V \log(V))$, or as an extension to the SIMD ISA of CPU and GPU. In our evaluation on CPU, when $V = 16$ (512-bit SIMD, 32-bit element), we achieve significant speedup on a range of representative kernels including set operations (8.4×), database joins (7.3×), sparse vector/matrix/tensor addition/multiplication on real/complex numbers (6.5×), merge sort (8.0× over scalar, 3.4× over the state-of-the-art SIMD), and SpGEMM (4.4× over the best one in the baseline collection).

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507728

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**.

KEYWORDS

SIMD, Key-value array, Sparse linear algebra, SpGEMM, Merge sort, Graph, Join

ACM Reference Format:
Bangyan Wang, Lei Deng, Fei Sun, Guohao Dai, Liu Liu, Yu Wang, and Yuan Xie. 2022. A One-for-All and $O(V \log(V))$-Cost Solution for Parallel Merge Style Operations on Sorted Key-Value Arrays. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507728

1 INTRODUCTION

There is a family of "merge style operations (MSO)" operations that appear frequently in different domains like scientific computing, deep learning, graph analytics, and database, and they often dominate the execution time. For example, sparse vector addition, a MSO, takes up to 98% execution time in SpGEMM (Figure 19). MSOs are very similar to merge sort but more general: they take two sorted key-value arrays as inputs and output another one, as

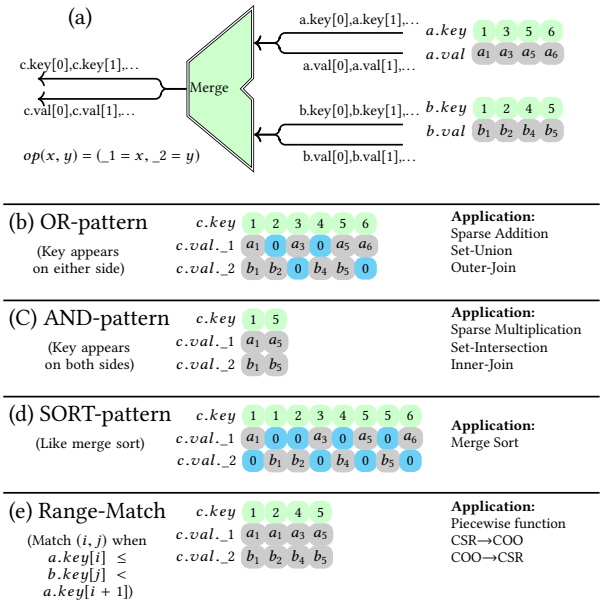


Figure 1: MSOs and the patterns.

Table 1: Different usage of the same pattern (OR-pattern) with different key/value types and calculations.

Problem	Key	ValA/ValB	ValC	$op(x, y)$
Sp Vec Add	int	float	float	$x + y$
Sp Mat Add	(int,int)	float	float	$x + y$
Sp Tens Add	(int,int,int)	float	float	$x + y$
Set-Union	int	-	-	-
Set-Union (idx+bitmap)	int	bitmap	bitmap	$x \| y$
Database Outer-Join (e.g.)	12byte string	city/name	(city,name)	(x, y)
Shortest Path	int	float	(int,float)	x<=y? ⌋ ("a",x):("b",y)

illustrated in Figure 1(a); they apply a binary function $op(x, y)$ to every pairs of values with matched keys, and probably remove, insert, or duplicate some elements when keys are matched or mismatched according to a certain rule called "pattern", like OR, AND, SORT, and Range-Match patterns as illustrated in Figure 1(b)-(e). How to accelerate MSOs has been extensively studied in different application domains. In scientific computing and deep learning, people usually implement the multiplication of two sparse matrices (SpGEMM) using MSOs with OR-pattern (row-wise or outer product) [10, 13, 21, 23, 24] or AND-pattern (inner product) [12, 16, 19, 22]. In graph analytics, the two most important operations are set intersection and difference [6], which can also be implemented using MSOs [2, 11]. In database, the join operation is one of the most expensive operations and the merge-based join is one of the extensively studied methods [5, 8, 14, 17]. Sorting is probably one of the most frequently used operation in all programs, which is also based on MSOs with SORT-pattern (merge sort) [8, 15, 17, 20]. Prior works have explored how to accelerate MSOs with SIMD-based CPU [5, 11, 14, 15, 17, 20], GPU [10, 20], and accelerators [2, 8, 12, 13, 16, 21–24]. Yet, these solutions suffer several major limitations:

(1) **The lack of a one-for-all solution for parallel execution**. How to execute MSOs in parallel with different "flavours" are solved as isolated problems using distinct methods. For example, the technique to find intersection (AND-pattern) [5, 11, 14] in parallel cannot be used to find union (OR-pattern) or merge sort (SORT-pattern). Most research fall into two patterns (AND and SORT) while many useful but less popular patterns like Difference, Left-Join, and Range-Match are rarely studied.

(2) **Incapability of supporting variable key/value data sizes and flexible calculations on matched pairs given fixed hardware**. Prior work on domain specific accelerators are based on monolithic hardware with tightly coupled datapaths for key-matching and value-calculation (e.g., +, ×) instead of a set of composable primitives. This restricts their capability to support variable key/value datatypes and value calculations as illustrated in Table 1. For example, sparse tensors and database tables may adopt 64-bit, 96-bit, 128-bit, or even larger key size (e.g., fixed size string of l bytes), while the hardware merge unit in prior work only supports fixed datatype (usually 32-bit for keys/values). This limits one merge accelerator to be used in only one merge application.

(3) $O(V^2)$ **expensive cost for given parallelism** V. Except for merge sort, all other parallel merge hardware [8, 16, 22, 24] or SIMD algorithms [5, 11, 14] are based on the $V \times V$ matrix comparator array, causing $O(V^2)$ resource cost for $O(V)$ performance [1] . As Jared et.al. point out in their work on database accelerator design [8], "*Because the number of comparators grows quadratically with the input width, it is difficult to implement hardware with a wide input array*".

In this work, we overcome above 3 challenges by proposing a ① **one-for-all** solution using a set of ② **composable primitives** with only ③ $O(V \log(V))$ **cost for O(V) performance** for MSOs. This proposed primitive can either be used as a SIMD instruction extension to the current CPU/GPU ISA, or be used as a drop-in replacement of the merge unit in prior domain specific accelerators[2, 8, 12, 13, 16, 21–24]. We perform performance evaluation on CPU platform. When the SIMD width V equals 16 (i.e., 512-bit), we achieve a significant speedup on a number of representative kernels including set operation (intersection, union, diff, etc.) (8.4×), database join (inner, left outer, full outer, etc.) (7.3×), sparse vector/matrix/tensor addition/multiplication on real/complex numbers (6.5×), merge sort (8.0× over scalar, 3.4× over the state-of-the-art SIMD), and SpGEMM (4.4× over the best one in the baseline collection).

In summary, this work makes the following contributions:

(1) **Formulation of the MSO concept** as a general and important target problem based on the observations of their applications in different domains including scientific computing, deep learning, graph analytics, database, and sorting.

[1] SpArch [24] suggests to reduce the number of comparators to $O(V^{3/2}) \sim O(V^{4/3})$ via a hierarchical array, but this number did not take into account the hardware cost of routing inputs into and concatenating results from low-level arrays.

(2) **Uniform representation of MSOs** based on the abstract sequential execution model "Restricted Zip Machine" (RZM). This model covers 2^{128} types of merge patterns including AND, OR, XOR, Diff, Left-Join, SORT, Range-Match, and even some useful but not yet discovered patterns.
(3) **Compiling method from any sequential RZM to parallel implementation with 4 proposed composable SIMD primitives** without the restriction on key/value datatype and value calculation ($op(x, y)$). The 4 primitives are `KeyCombine`, `Match`, `SEPermute` and `GetBreakpoint`.
(4) **$O(V \log(V))$-cost parallel hardware design to implement the primitives**. The latency is $O(\log(V))$ but can be fully pipelined.
(5) **Detailed evaluation** of the primitives on CPU by encoding them as SIMD instructions. It can achieve 4 ∼ 9 times speedup on a number of representative workloads.

The rest of this paper is organized as follows. Section-2 details the applications of MSOs. Section-3 explains the major challenges to implement MSOs in parallel and the related works. Section-4 introduces the abstract machine for sequential-to-parallel transformation. Section-5 introduces the 4 proposed primitives. In Section-6, we evaluate our design.

2 APPLICATIONS OF MSOS

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 with other operators and the value type is usually a data structure instead of a 32-bit float number, which can be implemented using the OR-pattern. In addition, graph mining can be implemented using set-intersection (AND-pattern) and set-difference (Diff-Pattern).

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

Database. Join operations combine the information of two tables into one table according to the matching keys. There are multiple flavors of join operations, as displayed in Figure 2. They can be implemented as MSOs with different patterns.

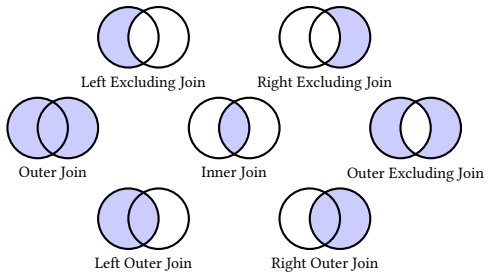


Figure 2: Various forms of join operations in database.

3 CHALLENGES AND RELATED WORKS

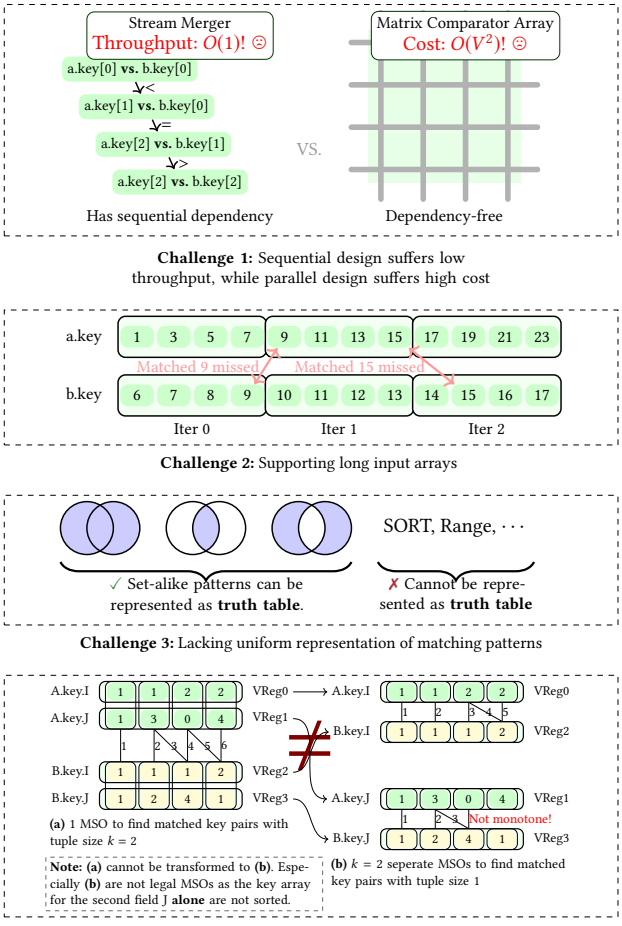


Figure 3: Challenges of MSOs.

Table 2: Latency of x86 all-to-all comparison instructions [3].

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

Table 3: Supported patterns using known techniques.

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

Sequential design suffers low throughput, while parallel design suffers high cost. An MSO can be implemented either in sequential or in parallel. The sequential design can be considered based on a FIFO merger that takes two FIFOs as input ports. In each cycle, the head of the two FIFOs are compared and the smaller one is popped. The advantages of sequential design are the simplicity and low cost, in addition, such FIFO merger can easily support different merge patterns such as AND-pattern [2, 9, 12] and OR-pattern [13, 21, 23]. The major limitations are its 1-tuple/cycle throughput caused by the sequential dependency to compare and pop keys in the FIFOs (as presented in the left of Figure 3-Challenge 1). To break the dependency of a later comparison on its previous comparison, parallel design usually performs all-to-all key comparison within a local scope of V elements. This method obtains V^2-bit comparison, resulting the a higher resource cost, as depicted in the right of Figure 3-Challenge 1. Accelerators usually face a trade-off between the $O(V^2)$ cost and the $O(1)$ performance [8, 16, 24]. On SIMD-based CPU, some works [5] use specialized instructions provided by x86 for all-to-all comparison but those instructions suffer very long latency (Table-2). Other works use last-byte-check and shuffle to equivalently achieve $\sqrt{V} \times \sqrt{V}$ comparison [11, 14]. After obtaining the V^2-bit result, key-matching is generated using specialized circuits (in accelerators) or using pre-computed look up tables. For intersection (AND-pattern), the table needs 2^V entries [5]; while for other patterns (e.g., OR-pattern), however, the table would require unacceptable 2^{V^2} entries, which is not applicable. For merge sort, both accelerators and SIMD implementation benefit from the sorting network technique introduced by Ken Batcher in 1968 [4] that only needs $O(V \log(V))$ comparison, however, this method is only applicable to SORT-pattern. Currently known supported patterns using sequential or parallel design are summarized in Table 3.

Support long input arrays. For hardware, the width V for parallel execution (e.g., SIMD width on CPU) is usually fixed (e.g., 4, 8, or 16), while the two input key-value arrays can be arbitrarily long. This problem can be solved by performing loop-tiling to divide the original problem into many sub-problems with shorter length. However, MSOs' loop-tiling is a more complicated problem than its element-wise counterpart, and does not have a general solution yet. First of all, the naive loop tiling cannot hold for almost 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 3-Challenge 2 gives an example on how matched elements get missed (9 and 15) if we naively tile the 12-elements input arrays into 3 sub-problems ([0:4],[4:8],[8:12]). Partial solutions indeed exist for AND-pattern and SORT-pattern. For AND-pattern, a widely adopted method is to compare the last element of the two input vectors and advance the smaller side [5, 8, 11, 14, 16]. For SORT-pattern, a method is to leave the larger half V elements of sorted result $2V$ elements to the next iteration and sort with the V elements from the smaller side of two input arrays[8, 15]. However, they are conditionally correct (not supporting the input with duplicated keys), work-inefficient (wasting 50% of useful works), and most importantly cannot generalize to other MSO patterns (e.g., OR-pattern). Unfortunately, there are no general solutions that work for all matching patterns known in literature.

The lack of uniform representation of matching patterns. There are no strict models that properly encode different patterns and handle them uniformly (see Figure 3-Challenge 3). Truth table is a good candidate to cover set-operation-related patterns (e.g., those in Figure 2), but it cannot cover SORT, Range-Match, or other one-to-many-match patterns. Most prior work focus on only one or two specific patterns and does not touch this problem.

Support variable tuple sizes via decomposition. Many applications of MSOs require the key and value to be tuples of multiple 32-bit fields. Because hardware is fixed, a single SIMD operation cannot directly support variable key tuple sizes. The only way is to decompose an MSO whose key tuple has k 32-bit fields into k fixed-sized operations. However the challenge is that one MSO with key tuple size k is very different from k independent MSOs with key tuple size of one and each MSO processes the one field of the original problem, as shown in Figure 3-Challenge 4, which blocks this way. For this reason, the same challenge is imposed to programmable platforms like CPU/GPU as well instead of just accelerators — the situation here very different from element-wise operations that complicated element-wise operation can be easily decomposed a finite set of basic element-wise operation like +,−,×, <, etc. Currently no prior work support variable key sizes except for a CGRA-based design [9] that can change the hardware datapath at runtime.

Discussion: Bandwidth and data reuse rate. A single MSO operation alone has relatively low arithmetic intensity which implies with the advance in the computation part of MSO, we also need to provide more higher-level data reuse or more memory bandwidth. For applications using MSO as basic building block, there are usually plenty of room for data reused between multiple MSOs, such as using merge tree buffered on-chip to implement multi-way merge [15, 24] or exploiting application specific data reuse [16]. Alternatively, accelerators often consider using emerging technologies like HBM to provide higher memory bandwidth. This paper is focused on the computation part of MSO, while exploiting higher level data reuse for each specific application can be achived at software level and is not the major focus of this paper.

4 UNIFORM REPRESENTATION

This section targets the challenges of finding a uniform representation to encode the matching patterns by introducing two sequential models: the general zip machine (GZM) and the restricted zip machine (RZM). GZM is a more general but strictly sequential machine. After partially specializing the state machine inside the GZM, we obtain RZM. Any RZM can be implemented with the proposed SIMD primitives, which will be described in Section 5.

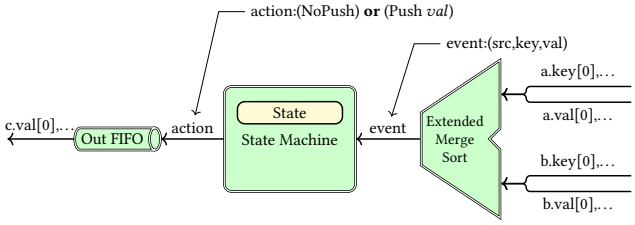


Figure 4: General zip machine (GZM).

4.1 General Zip Machine (GZM)

MSO represented in Figure 1(a) can be rewritten as "**a 2-to-1 merge sort followed by a state machine**", as illustrated in Figure 4. The idea behind this transform is that identical keys will become neighbours after 2-to-1 merge sort. This makes key matching becomes easier. Therefore, the very first stage of GZM is an "extended merge sort unit" that merges sort two sorted key-value arrays, besides, it record for each output key-value pair the input array it originally belongs to. Therefore its output is a sequence of tuples (src, key, val) where src can be 'a' or 'b' (1-bit), and we call each such tuple as an **event**. Then, the events will be sent to a state machine that executes events in sequential. The most usual way to use "state" is to memorize a sliding window of the event stream so that matched keys appear in the window will be found. For each new event, the state machine generates an **action**. The action can be either telling the "Out FIFO" to append a value *val* at the end (PushResult *val*), or doing no thing (NoPush). **Any MSO can be represented as a GZM by selecting an appropriate state machine**. For example, "sparse vector multiplication" can use a state machine that remember a sliding window of 2 events; if their *key*s match, then multiply their *val* part ($op(x, y) = x \times y$) and push the result. If their *key*s mismatch, then push nothing (NoPush). Other MSOs like "sparse vector addition"and "set-diff" have their respective state machines. For one-to-one matching, the state is usually a sliding window of 2 or 3 events. For one-to-many match, a key may duplicate arbitrary times and a infinite long sliding window seems necessary, but we can work around it by encoding extra information in the state while keeping the sliding window finite in the state machine.

Notice that the key ($c.key[0], \ldots$) is missing from the output array in Figure 4 (you only see $c.val[0], \ldots$) compared with Figure 1(a). This is an abbreviation without losing the generality: we can always obtain the key by constructing another proxy GZM where the original key is part of the proxy value. For example, we can let $ValC' = (Key, ValC)$, $ValA' = (Key, ValA)$, $ValB' = (Key, ValB)$, $op'((kx, vx), (ky, vy)) = ((kx|ky), op(vx, vy))$ in the proxy GZM to simultaneously obtain Key and $ValC$ from $ValC'$.

4.2 Restricted Zip Machine (RZM)

RZM is a modified from of GZM by restricting the state machine in GZM into a fixed pipeline with fewer parts being configurable. Yet RZM is still strong enough to cover interested use cases such as OR-pattern, SORT-pattern, etc. We first present several key ideas before introducing the formal definition.

Build two operand streams: The state machine in GZM transforms a stream (the **event** stream) into an output stream (the $c.val[0], \ldots$). This step can always be divided to two steps. 1) build two **operand** streams from the **event** stream that reflect the two operands of the binary operator $op(x, y)$. 2) apply operator $op(x, y)$ to the two **operand** streams in an element-wise fashion to produce the output stream. Figure-5 shows an example of set-union operation for input $a = [1, 2, 5, 8]$ and $b = [1, 3, 5, 7]$, where the two **operand** streams have all matched elements perfectly aligned with extra zeros inserted to proper places. Then we apply element-wise bitwise-OR to the two streams produce the union of sets $c.val = [1, 2, 3, 5, 7, 8]$. (The key and value for each event in the **event** stream are identical in this example so we present them in only one row to save some space in Figure-5 and latter figures. The zeros in blue are inserted while the rest comes from the value part of the **event** stream.)

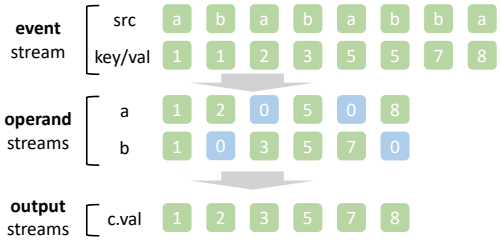


Figure 5: Operand streams as an intermediate step

Use FIFO and commands to build operand streams: Each **operand** stream can be built using a FIFO by accepting a command stream associated to the **event** stream. We define are four types of commands as shown in Table-4. By selecting appropriate commands we can align data in different ways that correspond to the desired matching pattern. Figure-6 shows an example to produce the **operand** streams in above set-union example. Figure-7 gives an extra example showing how *PushMostRecent* can helps implementing one-to-many matching patterns.

Table 4: 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)

The commands can be determined using the local information of slide-window of 3 events: The two command streams

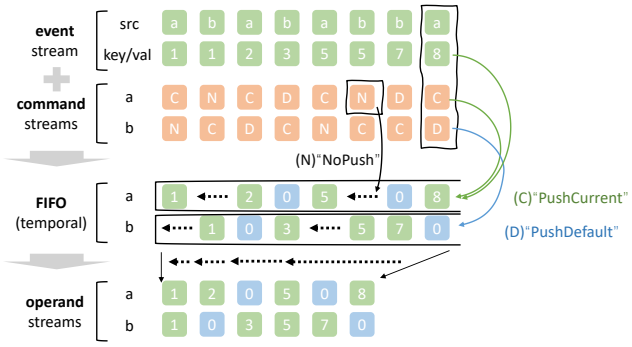


Figure 6: Use FIFO and command streams to build operand streams

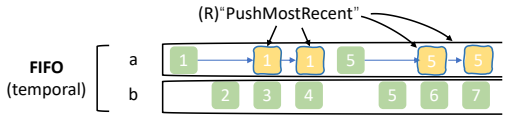


Figure 7: Extra example: Use *PushMostRecent* to implement one-to-many range-match

are also generated from **event** stream. For each **event**, we can decide its commands by only looking at its two neighbor events (left and right) in addition to the **event** itself. This is because the commands solely depend on whether a matched pair exists for given **event**, while after merge sort, checking two neighbors is sufficient to know if such matched pair exists.

Any matching pattern can be encoded as a function mapping 5-bit to 4bit: In the sliding window of 3 events (say **event[i-1**, **event[i]** and **event[i+1]**), each event is a tuple of (src, key, val), but there are only 5-bit information used as illustrated in Figure-8. The **val** part and the absolute value of **key** are not used to determine matching. Only the **src** of the 3 events and comparison relation ($<, =, >$) of **keys** matters. Further, because keys are already non-decreasing in the **event** stream, there are only two places where results can vary about the comparison: **event[i-1].key** =? **event[i].key**, and **event[i].key** =? **event[i+1].key**. Different matching patterns only differ on what two commands are generated for the 2^5 cases, so they can be encoded as function mapping 5-bit to 4-bit, or equivalently 128-bit ($2^5 \times (2+2)$ bits). In the rest of this paper, we name the stream of "5-bit"s as **info** stream. One such example of encoding range-match pattern into a such 5-bit to 4-bit function and its usage is shown in Figure-9[2].

Put everything together: The model of RZM therefore can be defined as like Figure-10: it starts by merging two input stream into a **event** stream like GZM. Then two command streams ($actions.a/b$) are generated by applying the function (encoding the pattern) to the 5-bit of each slide window of 3 elements. Next, two **operand** streams are built using two FIFOs (FIFO A/B), the **event** stream and two command streams. Finally, we drain the pair of FIFOs when both are non-empty and apply operator $op(x, y)$ to the pair

[2]In this example, we still let the key and value in input being identical for simplification. In practice values are usually different from keys.

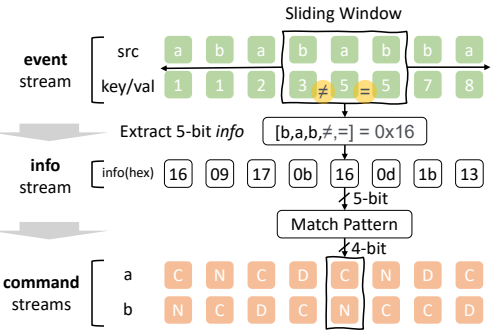


Figure 8: Extracting 5-bit info from the sliding window [pattern: union pattern (one-to-one)]

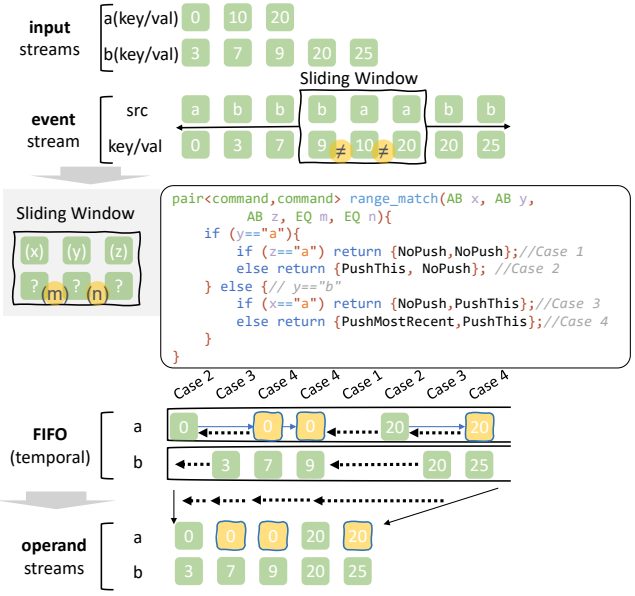


Figure 9: Example: Use range-match pattern to put b=[3,7,9,20,25] into bins [0,10),[10,20),[20,+∞) defined using delimiters a=[0,10,20]. The 5-bit are named as [x,y,z,m,n] respectively.
[pattern: range-match pattern (one-to-many)]

of popped elements and append the result to $c.val$, or do nothing otherwise.

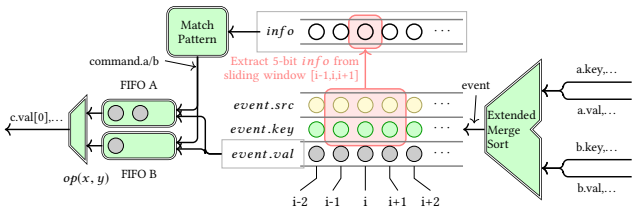


Figure 10: Restricted zip machine (RZM).

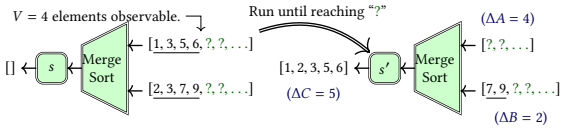


Figure 11: The imaginary execution

4.3 Divide-and-Conquer Relation

We discuss the divide-and-conquer relation of GZM (which also covers RZM as its special case), i.e. whether a GZM operation with long input arrays can be divided into smaller sub-problems with bounded size. This issues is important for using a hardware with fixed length to support input arrays with arbitrary length. We have already shown in Section-3 that naive tiling produce incorrect results. But we obtain following result: for any GZM, any length V (e.g. SIMD vector length), any initial state of its state machine s, and any two input arrays a and b:

$$\begin{aligned} &\exists s', \Delta A, \Delta B, \Delta C \quad \text{such that :} \\ &GZM(s, a, b) = GZM(s, a[0:V], b[0:V])[0:\Delta C], \\ &\quad + + GZM(s', a[\Delta A : end], b[\Delta B : end]) \end{aligned} \tag{1}$$

where ++ is array concatenation. Notice that the first term on the right side has bounded input, and the second term can continue to apply this division recursively until all sub-problems are bounded by size V.

To see why this relation hold, we can imagine executing $GZM(s, a, b)$ in following way: we first use paper cards with "?" on it to cover and hide the numbers of array a and b starting from their $V+1$-th elements, as shown in Figure-11-left. We run the GZM until it meet the first "?" so it can not continue execution. We set this timestamp as a breakpoint and record the number of elements consumed in two input arrays as ΔA and ΔB; the number of elements generated to the output as ΔC; and the current state of state machine as s', as shown in Figure-11-right. Then, we remove the all paper cards and resume execution until finished. Clearly, adding the cards and removing them will not change the output of GZM except dividing the execution into two halves. About the first half, the following two results are indistinguishable:

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

And about the second half, the following two results are indistinguishable:

(1) $GZM(s', a[\Delta A : end], b[\Delta B : end])$.
(2) The subarray of $GZM(s, a, b)$ starting from the $(\Delta C+1)$-th elements until the end.

Putting the two halves together, we obtain Equation-1.

5 FOUR PRIMITIVES TO IMPLEMENT RZM

We propose 4 vectorized primitives to implement RZM. They are `KeyCombine`, `Match`, `SEPermute`, and `GetBreakpoint`. Figure-12 compares the dataflow of RZM (a) and the implementation based on the four primitives (b). They will finish a RZM execution within a single pass: consume the two input streams a and b and generate output stream $c.val$ simultaneously. The intermediate stream defined in RZM model (the **event** stream, 2 **command** streams and 2 **operand** streams) are also consumed immediately after produced. Those streams are consumed/produced at vector granularity, i.e. $\sim V$ elements at an iteration. The RZM model is mapped to those primitives like following (at a first level approximation):

(1) Generating **event** stream from two input streams is finished by `KeyCombine`. It is basically a merge sort.
(2) Generating the two **command** streams from the **event** stream according to given matching pattern (encoded in 128-bit) is finished by `Match`. Its essence is executing multiple "32-entry 4-bit/entry" table lookups independently and in parallel.
(3) Generating the two **operand** streams from the **event** stream and **command** streams are finished by `SEPermute`. It is conceptually similar to a `permute/shuffle` instruction.
(4) Generating the **output** stream ($c.val$) from **operand** streams using binary operator $op(x, y)$ can be finished by traditional element-wise SIMD arithmetic primitives (for CPU/GPU) and don not require new primitives.
(5) Generating the divide-and-conquer parameter as mentioned in Section-4-3 (s', ΔA, ΔB and ΔC) is finished by `GetBreakpoint`.

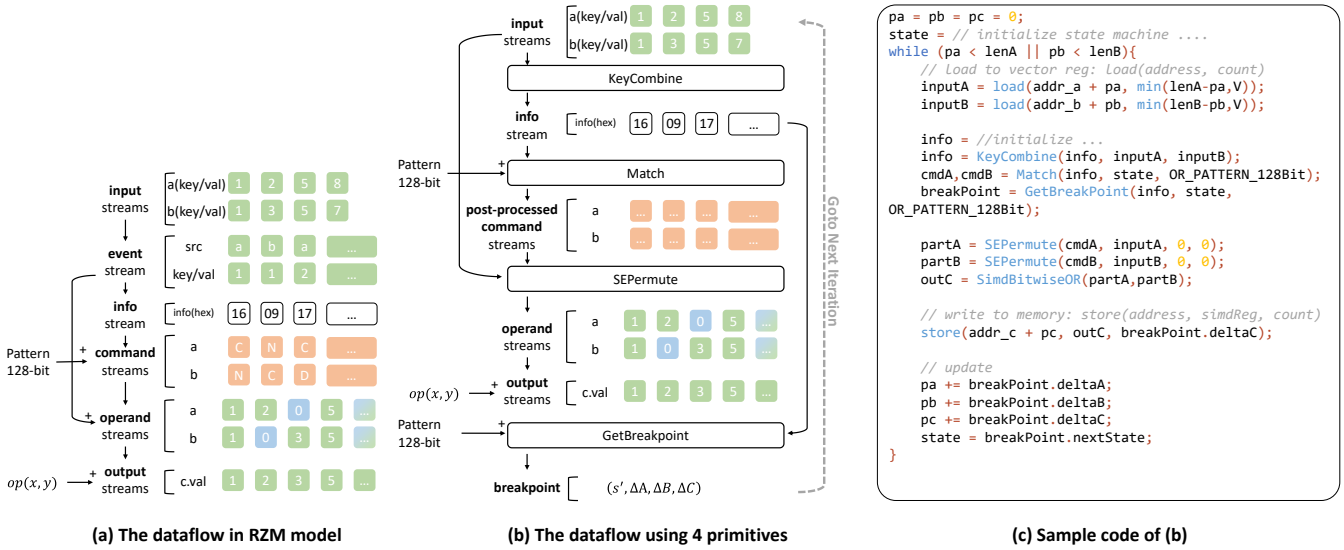
Above design is a plausible choice but has two limitations. First, we can bypass explicitly constructing the full **event** stream to improve efficiency. To see why, **command** generation in `Match` operation only use 5-bit for each sliding window of the **event** stream. And we can ask `SEPermute` to directly generate **output** stream from the **input** streams without relying **event** stream. So there is no reason to build the full **event** stream anymore. Second, in `SEPermute`, the **command** streams need extra preprocessing before being used by its permutation circuit. Because `SEPermute` will usually be invoked multiple times with the identical **command**, it will be more efficient to move this repeated work of preprocessing step from `SEPermute` to its upstream operation `Match` as a postprocesings step, as latter is only invoked once per iteration. Therefore, our practical design make extra optimizations to the earlier described one as following:

(1) `KeyCombine` will not produce a full **event** stream as output, but the **info** stream instead.
(2) `SEPermute` will directly read **input** streams instead of **event** stream.
(3) `Match` will take the **info** stream as input instead of **event** stream. `Match` also has an extra work: to postprocessing the **command** stream so that it can be directly used by the permutation circuit of **SEPermute**.

5.1 KeyCombine

`KeyCombine` takes the two **input** streams and merge sort them to make the **event** stream internally, followed by extracting the "5-bit"s for each sliding-window which include extracting the src of events (whether it is from 'a' or 'b') and compare whether keys of neighbors pairs are equal.

Use bitonic sorting network: A matured solution for merge sorting two sorted arrays (say a and b) with fixed length V (the hardware vector length) in parallel is to use the bitonic sorting



(a) The dataflow in RZM model (b) The dataflow using 4 primitives (c) Sample code of (b)

Figure 12: The correspondence between RZM model and the implementation using four primitives

network. One problem of the bitonic sorting network is its classical design can not ensure that the result is stable: i.e. when the elements' keys are the same, the results are sorted first according to their src (a's elements first, then b's) and then according to their original order in the input array. Stable property is required by our GZM/RZM model. A simple fix is to also encode the src and their location in original stream as part of the proxy key for comparison. For example, the second 2 in input array $a = [1, 2, 2, 3]$ would be encoded in an proxy key as:

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

Lexicographical sorting when every key is a tuple: Applications may require the keys to be tuples of multiple fields. Using classical bitonic sorting network to process each fields one-by-one will obtain their independent results instead of an lexicographically sorted array. We fixed this by passing the comparison results between the processing of different fields. From programmer's perspective, merge sorting tuples with k fields can be achieved using k `KeyCombine` invocations:

```
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)
```

where field1 is the most significant field and fieldk is the least significant field.

The underlying implementation is still based on bitonic sorting network but we argument each switcher with 2-bit state. Recall that classic bitonic sorting network is $\log(2V)$-layer butterfly-like network with each crossing being a 2-input 2-output compare-and-swap switcher. There are totally $V \log(2V)$ such switcher and each behave like a tiny sorter that only handle 2 elements. Now, the added 2-bit state allows each switcher to remember earlier result. If all earlier fields (those more significant fields) of the key are **equal**, the switcher is in **undecided** states and the decision will be made by latter (less significant) fields. Otherwise, the switch is in **frozen** states and latter fields cannot overwrite the comparison result anymore. We define 4 states using the 2-bits: Strong <, Strong > (the two **frozen** states), and, Weak <, Weak > (the two **undecided** states). For each `KeyCombine` invocation, each switcher has two proxy keys as input (x and y) and update its state fallowing:

(1) If the state is Strong < or >, it does not change.
(2) If the state is Weak < or >, and $x.key \neq y.key$, the state update to the Strong < or > depending whether $x.key <$ or $> y.key$.
(3) If the state is Weak < or >, and $x.key = y.key$, the state update to the Weak < or > depending whether $\langle x.src, x.loc \rangle <$ or $> \langle y.src, y.loc \rangle$.

All other aspects are the same as classic bitonic network. The switcher's state are also packed into `info`. In summery, `info` now contains following contents: 1) the 2-bit states for each one of the $V \log(2V)$ switchers, 2) the "5-bit"s collected from each sliding window.

Finally, after lexicographical merge sort, `KeyCombine` need to extract "5-bit" from each sliding window. The "src" can be taken from the sorted proxy keys of the last `KeyCombine` invocation, while when comparing a key with its neighbors' key, it need to accumulate the "equal" using logic AND of all past k `KeyCombine` invocations, since two keys are not equal if any one if their k fields is not equal.

5.2 SEPermute

`SEPermute` is used to produce the **operand** streams from the **input** stream. Its semantic is similar to the data permute/shuffle instruction and the post-processed **command** streams play similar roles like the control mask in conventional permute/shuffle instruction.

Only requires permutation that preserve order: The permutation required in this problem is restricted to those whose elements in output appears according to their original order of input. In other word, only removing elements, inserting constant elements (like 0), and duplicating elements are required, while exchanging order

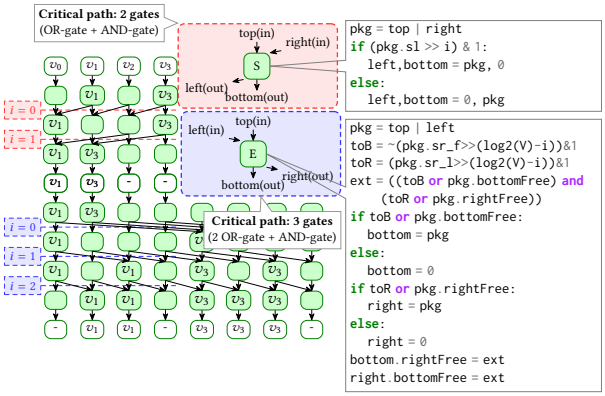


Figure 13: The squeeze and expanded network.

is never required. For example, for input [1,2,3,4], output could be [1,0,0,2,2,2,0,4] but never [4,2,3,1]. This implies the common $O(V^2)$-cost crossbar is overkill.

Use Squeeze-Expand Network: We introduce the Squeeze-Expand network with $O(V\log(V))$-cost to implement order-preserving permutation. It consists of $\log(V)$ squeeze layers followed $\log(2V)$ expand layers, as shown in Figure-13.

The squeeze network can drop part of the elements and shift the remaining elements towards left and 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 move straight down. Such usage of the squeeze network is not a new idea and was very commonly used for stream compaction.

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

The squeeze network + expand network is sufficient to cover all order preserving permutations. The key problem is to implement routing control logic. We first define the format of control signals (the post-processed **command** stream) it received from `Match`:

(1) For each element, there is a valid bit decide whether it will appear in the output.
(2) For each element, there is number specifying how many positions it will shift left in the squeeze network in total. It is a $\log(V)$ bit integer, named as `sl` ("**s**hift **l**eft").
(3) For each element, it will shift right and be broadcasted into a range in the expand network. There are two numbers specifying amount of right-shift for the first element and last element of the broadcasted range. If there is no duplication made, then the numbers for the first and the last will be equal. The two numbers are $\log(2V)$ bit integers, named as `sr_f` and `sr_l` ("**s**hift **r**right **f**irst/**l**ast").

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

Routing at squeeze network: At layer i, the squeeze network use the i-th bit ($i = 0$ is the least significant bit) of `sl` to decide whether it should shift left. The correctness is based on a simple fact: sl = sl[0]×2^0 + sl[1]×2^1 + sl[2]×2^2 + ... where sl[b] means `(sl>>b) & 1`).

Routing at expand network: At layer i, the expand network forwards elements towards right by $V/2^i$ positions or straight down or **both** (for duplication). Assume one copy of the element has been shift right by x positions (therefore `sr_f` $\le x \le$ `sr_l`). The routing rules are:

(1) Forward towards right if $x + V/2^i \le$ `sr_l`.
(2) Forward towards down if `sr_f` $\le x + 0$.

There is an efficient way to implement above rules but avoids using adders and comparison, which is favorable in hardware to reduce the cost and latency. As shown in Figure-13, we use two extra flag bits (`bottomFree` and `rightFree`) that initialized to 0 to incrementally build above inequality layer-by-layer and reduce the latency of each layer to 3-gates.

5.3 Match

`Match` first produces two **command** streams (for both a and b) from **info** stream followed by postprocessing it. The prior step is simply using each "5-bit" to lookup a 32-entry 4-bit/entry table encoding the matching pattern, which is embarrassingly parallel. The latter step is transforming the `command` streams into the (`valid`,`sl`,`sr_f`,`sr_l`) tuples as mentioned in `SEPermute`, which is in essence prefix sum operations. Following we show why. Because we have two streams a and b and they are symmetric, we denote them as ($x = a, \bar{x} = b$) or ($x = b, \bar{x} = a$). We name the postprocessed **command** streams as **postCmd** streams.

(1) (About `valid`) Whether an element is used in the final result depends only on if it is pushed, i.e. whether the command is *PushThis* in command stream x and src is x.
(2) (About `sl_l`) The number of positions to shift left for given element is the sum of prior elements that is not pushed to the array, i.e. the number of prior elements that their command is NOT *PushThis* in command stream x and src is x.
(3) (About `sr_f`) The number of positions to shift right for an element in x is the number of extra elements inserted before this element by either its opposite side $\bar{x}$, i.e. the number of prior elements whose command is NOT *NoPush* in command stream x and src is $\bar{x}$; or by self-side via inserting default elements, i.e. the number of prior elements whose command is *PushDefault* in command stream x and src is x.
(4) (About `sr_l`) The number of duplication (`sr_f`−`sr_l`) is the number of *PushMostRecent* executed to this element, i.e. the number of **directly** following elements that their command is *PushMostRecent* in command stream x, and not separated by any other commands like *PushThis* or *PushDefault*.

We notice that "counting the number of prior elements that satisfy certain condition" for all positions is a parallel prefix sum operation (which solved (2) and (3)), and similarly (4) is a parallel segmented suffix sum. Therefore deciding the tuple (`valid`, `sl`, `sr_f`, `sr_l`) for each element can be implemented in hardware using existing techniques. The only difference we need to take care of is that there are V results for a and V for b; the $2V$ results are mixed in merge sorted order. To unmix them, we can simply routing them along the reversed path in the bitonic network (the necessary

routing information is already known: the 2-bit states of comparators are packed into `info`). A further circuit level optimization is to merge the prefix sum operation into the unmixing step so they can be finished in one pass.

5.4 GetBreakpoint

The `GetBreakpoint` calculates the divide-and-conquer parameters discussed in Section-4.3, i.e. (s', ΔA, ΔB, ΔC). According to its definition, the first step is to decide when our GZM/RZM can not make any progress due to not knowing the value of the next element after the first V values of both input array. Realizing that that the last event executed before this breakpoint is one of the two V-th elements of two input arrays whichever appears earlier in the **event** stream, and therefore can be determined by only using the **event** stream. In our design, the `GetBreakpoint` takes the more compact **info** stream as input. To determine ΔA, ΔB, and ΔC, `GetBreakpoint` only need to count the number of consumed and pushed element according to the **command** stream until the last executed event before the breakpoint. Finally, properly encoding the state s' requires many engineering tricks and workarounds, which is omitted here but the main result is we managed to generate only two bits for s'.

6 EVALUATION

In this section, we evaluate the correctness, performance, and flexibility of our design. We implement a number of representative workloads that can be divided into two groups: the simple workloads that are direct MSOs and the complex workloads where an MSO is used as a basic building block of a more complicated operation. We compare the baseline CPU v.s. the same CPU with our SIMD extension, which is in essence comparing two optimized kernels that one only use traditional scalar and SIMD instructions while the other one also use the proposed SIMD primitives. For all workloads we cover, we use SIMD baselines when proper one can be found and otherwise scalar baselines (but as we mentioned in Section-3, only 2 special cases in MSO can be written in SIMD form using only existing SIMD primitives). We leave the other two use case of our design: "baseline GPU v.s. the same GPU + our SIMD extension", and "MSO-related accelerators v.s. the same accelerator + our $O(V\log(V))$-cost design as drop-in replacement" for future research.

6.1 Experiment Setup

We extend the GCC(v10.1.0)/Binutils(v2.35) toolchain and the gem5(v20.1.0.2) CPU simulator to support the proposed SIMD primitives. The base ISA is Armv8 with SVE extension. The simulated processor is an out-of-order CPU (the parameter sheet that models Arm Cortex A15). In addition, 32KB L1I cache, 32KB L1D cache, 1MB L2 cache, and a TaggedPrefetcher for L1D cache are enabled. Because the instruction latency in cycles is influenced by many factors such as the technology node, frequency, and voltage etc, we conservatively estimable the latency of the proposed primitives based on a simple reference: "32-bit integer addition can be performed in one cycle", and then set the cycle numbers of the instruction latency accordingly. To obtain an even more reliable lower bound of the performance improvement, we further scale the latency of each primitive by a factor of ~3, leaving sufficient margin for real implementation[3]. Finally, for given SIMD width V, the latency of the proposed primitives in the gem5 simulator is set as in Table 5: The area synthesized in 22nm is shown in Table-6. For reference, an intel 4 core Haswell processor at 22nm technology is about 200mm2, adding our primitives (with $V = 16$ for example) account for 0.098% area overhead.

Discussion: some practical engineering details: First, the encoded primitives require some bit-packing of their input/output to keep the number of input register operands below three and the number of output register being one. Second, although `KeyCombine` has relatively long latency, a sequence of cascaded `KeyCombine` primitives to handle multi-field keys can be fully pipelined even if there is data dependency on `info`, this is because different parts of `info` are updated in only one of the $\log(2V)$ layers of bitonic network and are updated in disjoint cycles, so cascaded `KeyCombine` primitives can be pipelined by layers.

Table 5: Latency of primitives.

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

Table 6: Area overhead 22nm

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

6.2 Simple Workloads

Figure 14 presents the throughput of set operations (set-union, set-intersection, set-diff, set-xor), database join operations (join-inner, join-outer, join-diff, join-xor, join-left). Those kernels demonstrate the flexibility to support different matching patterns of our method. Figure 15 presents the throughput of element-wise additions/multiplications for real/complex numbers between sparse vectors/matrices/tensors and an MSO related to the merge adjacency lists in graph analytics applications. Those kernels demonstrate the flexibility to support different tuple sizes for key and value, and different $op(x, y)$. All of them assume the inputs are already sorted as a key-value array. In our simulation, len(a)=len(b)=10k and the throughput is defined to be $(len(a) + len(b))/time$. We also label the speedup over the scalar[4] baseline for 3 of the SIMD width $V = 4, 16, 64$ (corresponding to SIMD bitwidth 128-bit, 512-bit, and 2048-bit). The average speedup for all above kernels when $V = 16$ is 7.1×.

6.3 Breakdown of Speedup

We analyze the origin of speedup by representing the execution time of an MSO as the product of three factors:

$$\text{Time(in cycles)} = \text{num of iter} \times \text{inst per iter} \times \text{cycs per inst.}$$

[3] If we do not scale the latency by this factor of 3, there will be ~20% extra performance improvement.

[4] Currently set-intersection is the only one among above kernels that has known method for SIMD implementation, so we only use scalar baseline here.

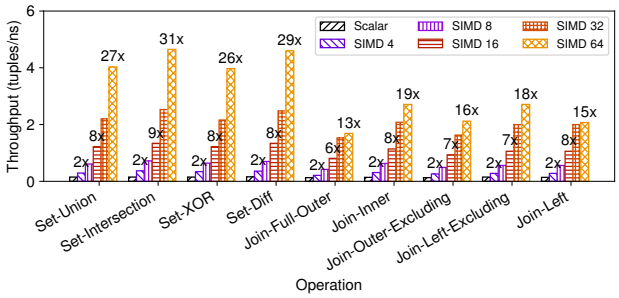


Figure 14: Throughput of set and join operations.

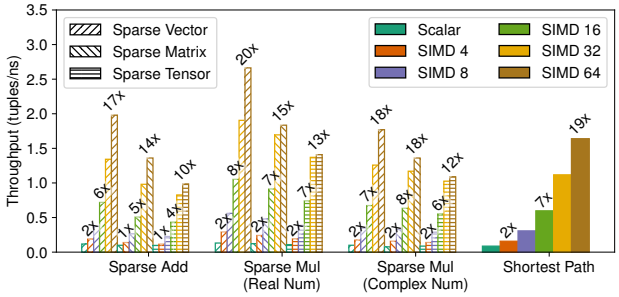


Figure 15: Throughput of sparse vector/matrix/tensor operations and graph operations.

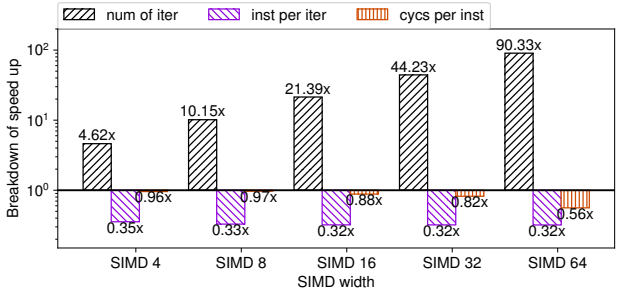


Figure 16: Breakdown of speedup of SIMD vs. scalar.

Figure 16 shows that how SIMD influences the three factors using sparse vector addition as example. We can observe that compared with the scalar implementation, the SIMD implementation enjoys (1) $\sim 1.5V$ times less iterations ☺, which is the main contributor of the throughput improvement, but suffers (2) ~ 3 times more instruction per loop ☹due to the increased complexity and (3) higher cycles-per-instruction (CPI) ☹caused by mixed factors. Specifically, the SIMD implementation eliminates the hard-to-predicate branch in scalar codes ☺but consumes longer instruction latency and is more vulnerable to bandwidth pressure ☹. The final speedup is the product of the three factors.

In general, the throughput scales well with the SIMD width V until $V = 32$. The throughput stops growing when V extends to 64 due to the saturated memory subsystem. Currently, the longest SIMD width on commodity CPU is 16 (AVX-512). So the saturation problem on $V = 64$ is only hypothetical.

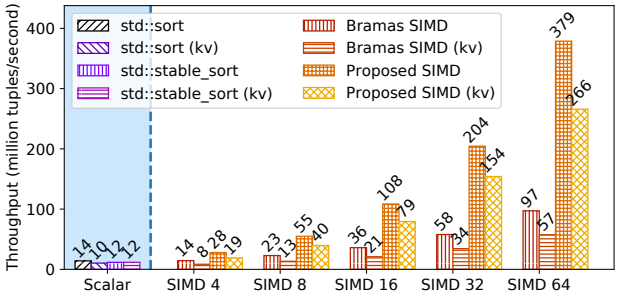


Figure 17: Throughput of sorting.

6.4 Complex Workloads

We evaluate two complex workloads that build on top of MSOs: 1) sorting, 2) general sparse matrix-matrix multiplication (SpGEMM).

Sorting. Our baseline includes two scalar implementations quicksort (`std::sort`), merge sort (`std::stable_sort`), and a SIMD implementation (Bramas SIMD[7]). The benchmark includes 32-bit interger sorting and 32-bit key-value sorting. The array is randomly generated and has 10k elements. Figure 17 shows the absolute performance of baselines and our design (the proposed SIMD). For $V = 16$, our method is 3.4× faster than Bramas SIMD and 8.0× faster than the scalar sorting algorithms.

Table 7: Properties of 4 sorting methods.

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

The proposed SIMD method is not only faster but also stable as shown in Table 7. Being stable means the relative order of key-value pairs with identical keys are preserved after sorting. Quicksort and existing SIMD sorting (bramas and others) algorithms are usually not stable. To stabilize their methods, a common fix is to pack the original location as a secondary key which introduces extra overhead, increasing the execution time by 1.5×[17].

`GetBreakpoint` primitives contribute two advantages over the naive loop-tilling scheme: (1) reducing the number of iterations by about a half, (2) ensuring the stable property. As Table 8 shown, the naive tilling method for sort is to sort $2V$ elements per iteration but only output V elements, while `GetBreakpoint` mechanism output about $1.78V$ elements per iteration on average when $V = 16$.

Table 8: The 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 is usually implemented as the weighed sum of sparse vectors and will benefit from faster sparse vector addition (which is an OR-pattern MSO) using SIMD. To evaluate the end-to-end performance improvement, we implement a SpGEMM kernel based on the SIMD-fied sparse vector addition and compare it with a collection of scalar baseline. Figure 18 depicts the absolute throughput as well as the the relative speedup for SIMD width 4/16/64 against the best one of the baseline collection on each dataset. For $V = 16$, it achieves 4.4× speedup on average. Figure 19 reports the percentage of the MSO in total SpGEMM execution time. The inner narrow rings represent different datasets and the outer wide rings represent their geometric mean. We can observe that the MSO dominates the execution time even with wide a SIMD unit (e.g., 90% at $V = 16$ (512-bit)) and the speedup of SIMD over the standalone sparse vector addition (as reported in Figure 15) is largely retained in end-to-end SpGEMM kernels. For $V = 32, 64$, the Amdahl's law provides slight effect because the non-MSO part (rest) consumes 14.5% ∼ 19% of the execution time.

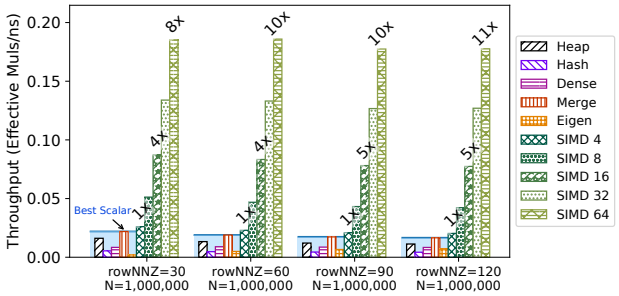


Figure 18: Throughput of SpGEMM and relative speedup of SIMD 4/156/64 vs. the best one of scalar.

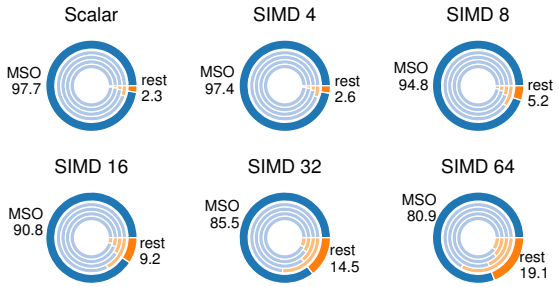


Figure 19: Percentage of execution time spent on the MSO in SpGEMM under different SIMD width.

We try our best to provide the high-quality SpGEMM baselines. We first consider the famous open source and portable library Eigen[1]. Because most well-optimized open source libraries and research artifact can only run on x86 while our platform is on Armv8, we cannot directly run their codes on the same simulated Arm core as our design. To handle this problem, we implement all major SpGEMM algorithms learned from literature in a portable way, including heap, hash, dense vector, and iterative merging, and carefully optimize them. Then, we compile and run them in both platforms (x86 and Arm) to serve as a bridge for inter-platform comparison (see Figure 20). We observe that our baseline collection has very close performance compared with the most recent and state-of-the-art SpGEMM implementation (Yu's Hash and Yu's HashVec[18]) that already beat MKL and ealier works. Therefore we believe our design will achieve similar speedup over those x86 libraries based on this indirect comparison. Finally, we want to mention that Yu's HashVec also exploits SIMD but it only uses SIMD to do parallel hash table probing, so it will only enjoy marginal benefits from the longer SIMD width in the future.

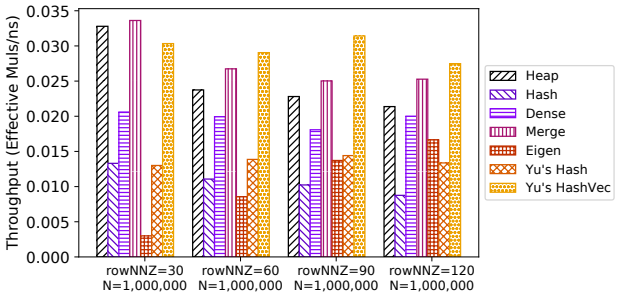


Figure 20: The portable baseline collection running on x86, compared with x86-only SpGEMM libraries (Yu's Hash and HashVec).

7 CONCLUSION

In this paper, we propose a one-for-all solution to implement MSOs in parallel and overcome the limitations of prior work. We point out that: (1) different MSO patterns can be encoded as functions that map a sliding window of 3 elements after merge sort to actions; (2) the V-to-V comparison problem appears in prior work can be transformed to a sorting problem; (3) a comparator with fixed size can be used to compare tuples of arbitrary size lexicographically, if only it is equipped with a 2-bit state machine. Our techniques allow a single piece of hardware to support 2^{128} patterns with the full flexibility in datatype and value calculation using only $O(V \log(V))$ hardware resource. In our evaluation on CPUs using the gem5 simulator, when $V = 16$ (512-bit SIMD, 32-bit element), we achieve significant speedup on a range of representative kernels including set operations (8.4×), database joins (7.3×), sparse vector/matrix/tensor addition/multiplication on real/complex numbers (6.5×), merge sort (8.0× over scalar, 3.4× over the state-of-the-art SIMD), and SpGEMM (4.4× over the best one in the baseline collection).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact include four components:

(1) **Modified GCC toolchain** that support the new instructions, which is available on https://github.com/AcrossForest/ModifedGCCForSIMDMerge.
(2) **Modified Gem5 CPU simulator** that support the new instructions, which is available on https://github.com/AcrossForest/ModifiedGem5ForSIMDMerge.
(3) **A set of kernel implementations** that exploiting the new instructions, which is available on https://github.com/AcrossForest/BenchmarksForSIMDMerge
(4) **A docker image/dockerfile** to provide out of the box expriment environment, which is available on (docker image) https://hub.docker.com/repository/docker/accross/simd$_{m}$erge or (dockerfile repo) https://github.com/AcrossForest/SIMDMergeDockerfile.

Alternatively, an archived repo of all above source code can be also found in Zenodo https://zenodo.org/record/5785310#.Ybry2mjMJPY.

The main focus of this artifact evaluation is the figures that directly compare the performance of baseline and proposed solutions, which is Figure-11,12,14,15.

A.2 Artifact Checklist

- **Algorithm:** SIMD sorted-set intersection, SIMD sorting, and SIMD SpGEMM.
- **Compilation:** GCC 10 or newer.
- **Run-time environment:** Ubuntu (docker image aviable).
- **Hardware:** Only CPU required.
- **Metrics:** Kernel execution time (or throughput).
- **Output:** The begin-to-finish execution time measured.
- **Experiments:** Run Gem5 CPU simulation.
- **How much disk space required (approximately)?:** 25GB (docker image, mainly the dependencies to build GCC/Gem5 from source).
- **How much time is needed to prepare workflow (approximately)?:** Below 10 minutes when using download docker images. Below 1 hour if building from scratch.
- **How much time is needed to complete experiments (approximately)?:** About one hour.
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** On Zenodo.

A.3 Description

Although building from the source code from github is viable, We recommend to pull docker images or build from the Dockerfile directly to avoid the dependency problems during environment setup.

A.4 Installation

You only need to pull the docker image.

```
docker pull accross/simd_merge:latest
```

A.5 Experiment Workflow

Run the docker image and login into the container like following.

```
docker run -it accross/simd_merge:latest /bin/bash
cd /root/
cat readme.md # <- follow the instructions here
```

There is a readme.md file in the /root/ folder. Follow the step-by-step instructions to run the scripts with appropriate parameters, which will invoke Gem5 simulator to run the kernels and print the execution time in the standard output. We need to collect those results, calculate the throughput and reproduce Figure 11,12,14 and 15.

A.6 Evaluation and Expected Results

We expect the performance results (especially the 3.4~8.4x speedup over the baseline) in Figure-14,15,17,18 to be accurately reproduced.

REFERENCES

[1] [n. d.]. 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.
[2] [n. d.]. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. ([n. d.]).
[3] [n. d.]. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.
[4] K. E. Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*. ACM Press, New York, New York, USA. https://doi.org/10.1145/1468075.1468121
[5] Benjamin Schlegel,Thomas Willhalm, Wolfgang Lehner. [n. d.]. Fast Sorted-Set Intersection using SIMD Instructions. ([n. d.]).
[6] Maciej Besta, zur Vonarburg-Shmaria, Yannick Schaffner, Leonardo Schwarz, Grzegorz Kwasniewski, Lukas Gianinazzi, Jakub Beranek, Kacper Janda, Tobias Holenstein, Sebastian Leisinger, Peter Tatkowski, Esref Ozdemir, Adrian Balla, Marcin Copik, Philipp Lindenberger, Pavel Kalvoda, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. [n. d.]. GraphMineSuite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra. https://arxiv.org/pdf/2103.03653
[7] Bérenger Bramas. 2021. A fast vectorized sorting implementation based on the ARM scalable vector extension (SVE). *ArXiv* abs/2105.07782 (2021).
[8] Jared Casper and Kunle Olukotun. 2014. Hardware acceleration of database operations. In *FPGA '14*. ACM, New York. https://doi.org/10.1145/2554688.2554787
[9] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 10122019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, USA, 924–939. https://doi.org/10.1145/3352460.3358276
[10] Felix Gremse, Andreas Höfter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. 2015. GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging. *SIAM Journal on Scientific Computing* 37, 1 (2015), C54–C71. https://doi.org/10.1137/130948811
[11] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 05272018. Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions. In *Proceedings of the 2018 International Conference on Management of Data*, Gautam Das, Christopher Jermaine, and Philip Bernstein (Eds.). ACM, New York, NY, USA, 1587–1602. https://doi.org/10.1145/3183713.3196924
[12] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 10122019. ExTensor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275
[13] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2/27/2021 - 3/3/2021. SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 84–96. https://doi.org/10.1109/HPCA51647.2021.00017
[14] Hiroshi Inoue. [n. d.]. Faster Set Intersection with SIMD instructions by Reducing Branch Mispredictions. ([n. d.]).
[15] Hiroshi Inoue and Kenjiro Taura. 2015. SIMD- and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1274–1285. https://doi.org/10.14778/2809974.2809988

[16] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, and S. W. Keckler and Z. Zhang. [n. d.]. C24-4 SNAP: A 1.67 – 21.55TOPS/W Sparse Neural Acceleration Processor for Unstructured Sparse Deep Neural Network Inference in 16nm CMOS. ([n. d.]).

[17] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash revisited. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389. https://doi.org/10.14778/1687553.1687564

[18] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 08132018. High-Performance Sparse Matrix-Matrix Products on Intel KNL and Multicore Architectures. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3229710.3229720

[19] Julian Pavon, Ivan Vargas Valdivieso, Adrian Barredo, Joan Marimon, Miquel Moreto, Francesc Moll, Osman Unsal, Mateo Valero, and Adrian Cristal. 2/27/2021 - 3/3/2021. VIA: A Smart Scratchpad for Vector Units with Application to Sparse Matrix Computations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 921–934. https://doi.org/10.1109/HPCA51647.2021.00081

[20] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 06062010. Fast sort on CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, Ahmed Elmagarmid and Divyakant Agrawal (Eds.). ACM, New York, NY, USA, 351–362. https://doi.org/10.1145/1807167.1807207

[21] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 10/17/2020 - 10/21/2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780. https://doi.org/10.1109/MICRO50266.2020.00068

[22] Sungju Ryu, Youngtaek Oh, Taesu Kim, Daehyun Ahn, and Jae-Joon Kim. [n. d.]. SPRITE: Sparsity-Aware Neural Processing Unit with Constant Probability of Index-Matching. ([n. d.]).

[23] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 04192021. 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*, Tim Sherwood, Emery Berger, and Christos Kozyrakis (Eds.). ACM, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702

[24] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. [n. d.]. SpArch: Efficient Architecture for Sparse Matrix Multiplication. https://arxiv.org/pdf/2002.08947