FEASTA: A Flexible and Efficient Accelerator for Sparse Tensor Algebra in Machine Learning

Kai Zhong
Tsinghua University, China
zhongk19@mails.tsinghua.edu.cn

Zhenhua Zhu
Tsinghua University, China
zhuzhenhua@mail.tsinghua.edu.cn

Guohao Dai*
Shanghai Jiao Tong University, Infinigence-AI, China
daiguohao@sjtu.edu.cn

Hongyi Wang
Tsinghua University, China
wanghong22@mails.tsinghua.edu.cn

Xinhao Yang
Tsinghua University, China
yxh21@mails.tsinghua.edu.cn

Haoyu Zhang
Tsinghua University, China
hy-zhang22@mails.tsinghua.edu.cn

Jin Si†
Beijing University of Posts and Telecommunications, China
jagnate@bupt.edu.cn

Qiuli Mao
Tsinghua University, China
maoql21@mails.tsinghua.edu.cn

Shulin Zeng
Tsinghua University, Infinigence-AI, China
zengsl18@mails.tsinghua.edu.cn

Ke Hong
Tsinghua University, China
hongk21@mails.tsinghua.edu.cn

Genghan Zhang†
Stanford University, USA
zgh23@stanford.edu

Huazhong Yang
Tsinghua University, China
yanghz@tsinghua.edu.cn

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

Abstract

Recently, sparse tensor algebra (SpTA) plays an increasingly important role in machine learning. However, due to the unstructured sparsity of SpTA, the general-purpose processors (e.g., GPU and CPU) are inefficient because of the underutilized hardware resources. Sparse kernel accelerators are optimized for specific tasks. However, their dedicated processing units and data paths cannot effectively support other SpTA tasks with different dataflow and various sparsity, resulting in performance degradation. This paper proposes FEASTA, a Flexible and Efficient Accelerator for Sparse Tensor Algebra. To process general SpTA tasks with various sparsity efficiently, we design FEASTA meticulously from three levels. At the dataflow abstraction level, we apply the Einstein Summation on the sparse fiber tree data structure to model the unified execution flow of general SpTA as joining and merging the fiber tree. At the instruction set architecture (ISA) level, a general SpTA ISA is proposed based on the execution flow. It includes different types of instructions for dense and sparse data, achieving flexibility and efficiency at the instruction level. At the architecture level, an instruction-driven architecture consisting of configurable and high-performance function units is designed, supporting the flexible and efficient ISA. Evaluations show that FEASTA has 5.40× geomean energy efficiency improvements compared to GPU among various workloads. FEASTA delivers 1.47× and 3.19× higher performance on sparse matrix multiplication kernels compared to state-of-the-art sparse matrix accelerator and CPU extension. Across diverse kernels, FEASTA achieves 1.69-12.70× energy efficiency over existing architectures.

CCS Concepts: • **Hardware → Application specific instruction set processors**; *Hardware accelerators*; • **Computer systems organization** → Special purpose systems.

Keywords: Sparse Computation, Tensor Algebra, Instruction Set Architecture, Hardware Accelerator

ACM Reference Format:
Kai Zhong, Zhenhua Zhu, Guohao Dai, Hongyi Wang, Xinhao Yang, Haoyu Zhang, Jin Si, Qiuli Mao, Shulin Zeng, Ke Hong, Genghan Zhang, Huazhong Yang, and Yu Wang. 2024. FEASTA: A Flexible and Efficient Accelerator for Sparse Tensor Algebra in Machine Learning . In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume*

*They are both corresponding authors.
†They made contributions to this paper while at Tsinghua University.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651336

3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620666.3651336

1 Introduction

Recently, sparse tensor algebra (SpTA) has become increasingly important in the field of machine learning due to its potential for reducing the workload and processing naturally sparse data in various applications [10]. On the one hand, sparse pruning algorithms [16, 21, 51] are widely used in model compression, where they can significantly reduce the workload by setting part of parameters to zero. Similar methods have also been used to prune attention maps or feature maps, thus speeding up transformer models by reducing a large amount of computation [3, 13, 32, 43]. On the other hand, as the application field of deep learning is gradually expanding, sparse tensor data and sparse computations become natural requirements in some emerging applications. For example, the sparse point cloud data is commonly used in autonomous driving [6, 30] and processed by sparse convolution neural networks [8, 54]. The sparse graph data is widely used in fields like e-commerce [57], circuit design [55], and biochemistry [52], where graph neural networks (GNNs) achieve better accuracy than traditional algorithms [67].

Towards the wide range of application scenarios, the general SpTA tasks demonstrate diversity in terms of computational dataflow, sparsity, and data distribution. The dataflow of SpTA tasks is different in various operators, such as sparse convolution (SpConv), dense matrix multiplication (MM), sparse matrix-matrix multiplication (SpMM), sparse matrix-sparse matrix multiplication (SpMSpM), sampled dense and dense matrix multiplication (SDDMM) [4, 23], etc. Based on the specific application and input data, the density of general SpTA tasks varies from 0.01% [28] to 90% [62]. Both dense and compressed data formats are used to represent and process data with different levels of sparsity. In addition, the sparse data may exhibit an unbalanced distribution of non-zero values, which causes an unbalanced workload and brings new challenges to hardware performance [68].

General-purpose processors like CPU and GPU [5, 9] are usually ***inefficient*** when processing sparse tensors. At the hardware level, the tensor core and vector processing units aim at dense data. When dealing with unstructured sparse data, they waste computing resources because of processing zero values. At the programming model level, the general SIMT model focuses on computing regular data in parallel. However, the non-zero values of sparse data are irregularly distributed. The mismatch between the programming model and data leads to redundant branching, synchronizing overheads, and low resource utilization [24]. To improve the performance of SpTA on general-purpose processors, some CPU ISA extensions [42, 53] of stream data are proposed. However, their CPU-based architectures exhibit low stream parallelism, resulting in 2.4× performance degradation compared to specialized accelerators [42]. Some other sparse CPU architectures use many lightweight threads [1, 20]. However, they have no dedicated instructions, registers, and function units for sparse data, resulting in a large number of instructions for SpTA and prolonged processing time.

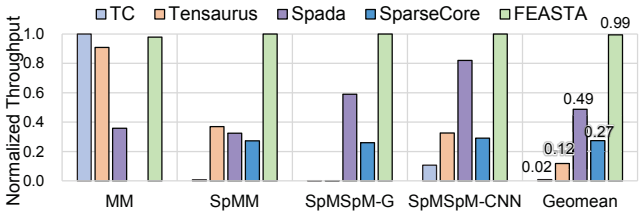


Figure 1. The throughput of different accelerators processing different SpTA kernels (normalized to the best).

To address the performance issue of SpTA tasks on general-purpose processors, researchers proposed sparse kernel accelerators that support irregular dataflow through dedicated data paths and computing units [31, 35, 40, 47–49, 63, 65]. However, they are optimized for specific sparse operators, making them ***inflexible*** and inefficient for other operators with various degrees of sparsity, as shown in Fig. 1. Take the SpMSpM accelerator Spada as an example. It has a fixed functional data path and has to process dense input with sparse data format, resulting in large overhead and low performance of SpMM and MM. Besides, existing general SpTA accelerators [19] can only support one iteration order, making its performance lower than other sparse kernel accelerators [48, 49, 65]. Some abstract models for streaming architectures [23, 60] can describe various dataflow, but designing a dedicated accelerator for each SpTA is also inflexible.

Motivated by the issues and challenges above, this paper proposes **FEASTA, a Flexible and Efficient Accelerator for SpTA**. FEASTA conducts optimizations from three aspects, i.e., data flow abstraction, ISA, and hardware architecture, as illustrated in Fig. 2. At the **dataflow abstraction** level, FEASTA applies the Einstein Summation (EinSum) formula to the Sparse Fiber Tree (SFT) data structure for describing general SpTA. The analysis shows that SpTA can be abstracted to a unified execution flow of the joining and merging of fiber trees. At the **ISA** level, a general SpTA ISA is proposed based on the unified execution flow. Seven basic types of instructions are defined to accomplish the execution flow. Each ISA type includes several specific instructions tailored for both dense and sparse data formats. Besides, instruction packing and multi-threading execution strategies are proposed to enable SpTA computation in parallel. Thus, both flexibility and efficiency are guaranteed at the instruction level. At the **architecture** level, an instruction-driven and dataflow configurable architecture is designed to support flexible and balanced computations.

The contributions of this paper are listed as follows:

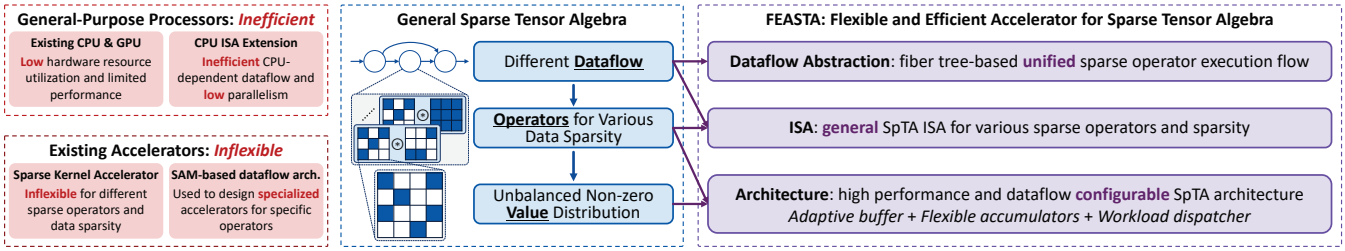


Figure 2. The challenges of existing SpTA work and our solutions of FEASTA.

- A unified execution flow abstraction for general SpTA is proposed. Various sparse operators can be modeled as iterative fiber tree join and merging.
- A general SpTA ISA is proposed to achieve high flexibility and computational efficiency at the instruction level. We design instructions for both sparse and dense data formats and enable different instruction parallelization methods.
- An instruction-driven architecture is proposed to support SpTA efficiently. The configurable multi-mode computation units can support tensor calculation in flexible ways. The dispatcher can also balance the workload during runtime.

FEASTA can effectively support various sparse operators, including SDDMM, that other sparse accelerators cannot process efficiently. Extensive experimental results show that FEASTA improves the sparse kernel throughput by 1.47× than SOTA accelerators and achieves 5.66× geomean power efficiency when running diverse SpTA workloads.

2 Preliminary

2.1 Sparse Tensor and Data Format

In this paper, we define concepts similar to TACO [26]. A *scalar* is a number, while an N-D *tensor* is an N-dimensional data array. Particularly, a 1-D tensor is a vector, and a 2-D tensor is a matrix. Each element in the N-D tensor is located by an integer N-tuple as *coordinates*. The coordinate of the i-*th* dimension is i-*crd*. Concatenated subscript A_{ijk} represents the element with the coordinate of (i, j, k) in tensor A, and the sizes of each dimension are I, J, K. Sparse tensors usually contain a large number of zero elements. We refer to the percentage of zero elements in a tensor as its *sparsity*. We call a tensor with a sparsity of $\sim 0\%$ a dense tensor.

In SpTA, tensors can be stored in two types of formats, i.e., dense data format and compressed data format. In the *dense data format*, all elements are stored sequentially, regardless of zero or non-zero. As all elements are stored in sequence, there is no need to store coordinates. If most elements in a tensor are zero, storing only the non-zero elements would be more space efficient, which is referred to as *compressed data format*. Besides the non-zero values, the coordinates of non-zeros also need to be encoded and stored. There are various common compressed data formats, including Compressed Sparse Row (CSR), Coordinate Format (COO), BitVector, etc. Fig. 3(b) and (c) show the examples of dense data format and compressed data format of a 2-D sparse tensor in Fig. 3(a).

2.2 Sparse Fiber Tree Data Structure

To avoid getting bogged down in the details of formats and to provide a unified description for sparse tensors, in this paper, we use the Sparse Fiber Tree (SFT) [50, 60] data structure. And we will show that it is also useful for dense tensor operating. The SFT (also called compressed sparse fibers) is a hierarchical multidimensional sparse tensor representation. Specifically, it recursively expands the sparse tensor along each dimension, dividing it into sub-tensors based on coordinates. SFT can represent both dense data format and compressed sparse data format, and the two examples are shown in Fig. 3(c) and (d). The 2-D sparse tensor of Fig. 3(a) contains two dimensions, i.e., i-dimension and j-dimension. When expanding the tensor in the order from i-dimension to j-dimension, the SFT consists of two levels of fibers, which are denoted as i-fiber and j-fiber. Each fiber in SFT contains two arrays: the coordinate (*crd*) array and the pointer (*ptr*) array. For the last level of SFT, the *ptr* array is replaced by a value (*val*) array. The *crd* array stores the coordinates of all sub-tensors (i.e., child-fiber) in this dimension. For example, the *crd* array of i-fiber in Fig. 3 contains the row coordinate of each j-fiber, while the *crd* array of j-fiber represents the column coordinate of each value. Fig. 3(d) and (e) give two examples of SFT representing dense tensor and sparse tensor.

2.3 Sparse Kernels in Deep Learning

Sparse tensors exist widely in various real-world machine learning applications. For brevity, we use two-dimensional matrix algebra as an example, and the discussion can be generalized to multidimensional tensors. For computer vision and natural language processing, the original input data and weights are dense. The operations (e.g., convolutions) on these dense data can be abstracted as the dense Matrix Multiplication (MM) operator. Weight pruning [16] is a classic model compression method that obtains sparse weights. The operations between pruned weights and dense features can be abstracted as the Sparse Matrix-Matrix Multiplication (SpMM) operator. When the features are also sparse,

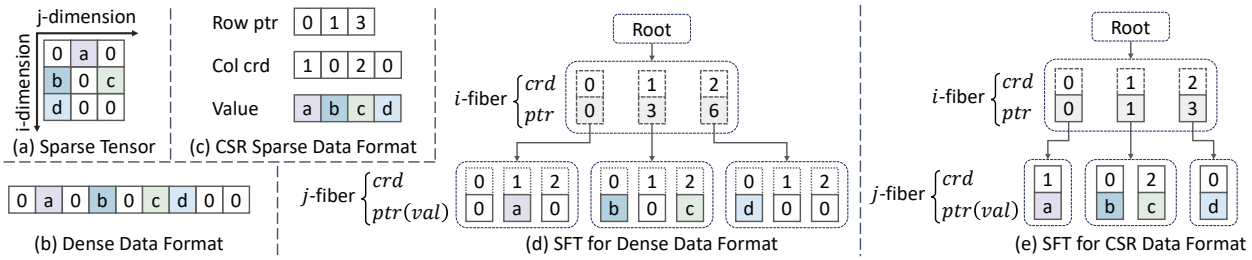


Figure 3. (a) A 2-D sparse tensor and its (b) dense format, (c) CSR format. (d) (e) are SFTs for the dense format and CSR format.

the Sparse Matrix-Sparse Matrix Multiplication (SpMSpM) operator is obtained. Pruning can also be applied to attention maps of Transformer models, resulting in the SpMM operator between attention maps and features [13, 43]. Considering the sparse mask of the attention map is known before the computation, it can be simplified to Sampled Dense-Dense Matrix Multiplication (SDDMM) to avoid unnecessary calculations [33]. Emerging algorithms are proposed to process the sparse input data. For example, the operation of aggregating features through edges in GNNs can be abstracted as the SpMM [58] or SpMSpM [59]. In point cloud neural networks, sparse 3D voxels are processed by sparse convolution [8].

In order to describe SpTA more concisely, we leverage the EinSum notation [2], which describes how tensor elements are indexed and multiplied and how results are summed over dimensions. For example, the EinSum notation for matrix multiplication is $C_{ij} = A_{ik}B_{kj}$. For each output C_{ij}, the required input elements (A_{ik} and B_{kj}) for computing it are indexed by coordinates and multiplied; then the result is obtained through reduction over dimension k.

2.4 Limitations of Existing Work and Our Goals

Existing kernel accelerators have insufficient flexibility due to their fixed functional units and data paths. On the one hand, the number of supported operators is limited. Sextans [47] is designed for SpMM; some other accelerators are designed for SpMSpM, and they can not support multidimensional tensors and SDDMM kernel. Tensaurus [49] can support several sparse-dense SpTA operators like SpMM and SpMTTKRP but cannot process SDDMM and SpMSpM in a sparse way. SAM [23] proposes basic data stream blocks to describe streaming dataflow accelerators for any SpTA, but it will generate dedicated data paths and inflexible dataflow-driven architectures for specific kernels.

Design Goal 1: *Support different multidimensional tensors and operators with unified hardware rather than designing dedicated accelerators for each SpTA task.*

On the other hand, existing sparse kernel accelerators also suffer from efficiency degradation when processing different sparsity tensors, as shown in Fig. 1. The tensor core (TC) architecture is optimized for MM kernels [7], thus have to read and calculate all elements including zeros when running SpMM and SpMSpM tasks, wasting computing resources. The Tensaurus [49] optimized for SpMM kernels faces the same problem when running SpMSpM tasks. For the architectures optimized for SpMSpM, e.g., Spada [31] as the SOTA, dense matrices need to be encoded in compressed sparse format. As a result, the redundant coordinate storage, access, and decoding cause severe performance degradation.

Design Goal 2: *Perform tailored processing on different sparsity and sparse/dense formats, avoiding redundant overheads.*

Towards flexible SpTA computation, multiple ISA and compiler studies have been conducted. SparseCore [42] and RZM [53] extend CPU ISA for stream operators. Though the flexibility can be guaranteed by the generality of CPU, the performance is limited by low parallelism of CPU and complex data paths designed for general ISA. SAM [23] can model and design dedicated streaming dataflow accelerators for different sparse kernels. But it lacks hardware-level performance-oriented optimizations, e.g., parallelization, workload balance, and memory hierarchy management.

Design Goal 3: *Define a domain-specific ISA for general SpTA. Enable instruction-driven flexible data path and memory management with efficient parallelization methods.*

3 SpTA Dataflow and FEASTA ISA

For the flexibility required in ***Design Goal 1*** and ***2***, we start from the SFT abstract data structure and apply the EinSum formula to it, getting a unified execution flow as traversal, join, and merging of subtrees on the SFT. This allows SpTA to be supported by some key sub-operators on fibers. We design the ISA to operate fibers, aiming at ***Design Goal 3***.

3.1 From EinSum to Execution Flow

Existing sparse kernel accelerators face limited versatility problems due to their specialized design for specific sparse kernels. To overcome it, we first represent all tensors with a unified SFT data structure. Then, the execution flow of general SpTA is modeled by applying the EinSum formula to the SFT data structure. SFT is a level-wise data structure, and each level has fibers of one tensor dimension. The EinSum formula defines the computation rules of SpTA by expressing the relationship of dimensions, i.e., the relationship of fibers in SFT. We analyze EinSum from the following aspects:

Table 1. FEASTA ISA. Each instruction has 32 bits: 4-bit OPCODE, 6-bit for each register, 3-bit TYPE, and 2-bit MOD.

Instruction	Description	Operands (Fd, Ft, and Fs are the ids of FR, Cs and Ct are the ids of CR)
load Fd Fs Cs TYPE	load fiber into FR	Fd:destination, Fs:beg&len, Cs:start address, TYPE=C/U/sv/dv
load-i Fd Cs Ct TYPE	load fiber into FR according to CR	Fd:destination, Cs:start address, Ct:length, TYPE=C/U/sv/dv
load-d Fd Fs Ct	calculate D fiber and save in FR	Fd:destination, Fs:beg&len, Ct:child-fiber length
load-di Fd Cs Ct	calculate D fiber and save in FR according to CR	Fd:destination, Cs:length, Ct:child-fiber length
read Fs Cs TYPE	read fibers into cache	Fs:beg&len, Cs:start address, TYPE=C/U/sv/dv
read-i Fd Cs Ct TYPE	read fibers into cache and save its beg&len	Fd:destination beg&len, Cs:start address, Cs:length, TYPE=C/U/sv/dv
inter Fd1 Fd2 Fs Ft TYPE	intersection two fibers and save results into FRs	Fd1:result1, Fd2:result2, Fs:fiber1, Ft:fiber2, TYPE=CC/CU/Csv/Cdv/Usv
inter-d Fd1 Fd1 Fs Ct	intersection of one fiber and a D fiber	Fd1:result1, Fd2:result2, Fs:fiber1, Ct: child-fiber length
dmr-e Fd Fs Ft TYPE DP MOD	calculation of element-wise fiber operations	Fd:result fiber, Fs:fiber1, Ft:fiber2, DP:data parallelism, MOD=inter/union
dmr-p Fd Fs Ft TYPE DP FP MOD	calculation of fibers products	Fd:psum fibers, Fs:fiber1, Ft:fiber(s)2, DP/FP:data/fiber parallelism, MOD=in/out/row
acc Fd MOD TYPE FP	accumulate psums	Fd:psum fibers, MOD=in/out/row, TYPE=sv/dv, FP:fiber parallelism
write Fd Fs Cs TYPE	write back fibers from cache into DRAM	Fd:result beg&len, Fs:beg&len, Cs:start address, TYPE=C/U/sv/dv
store Fd Fs Cs TYPE	store fiber from FR into DRAM	Fd:result beg&len, Fs:fiber Cs:start address, TYPE=C/U

(1) The relation of tensor indices on the right side defines the iteration space of SpTA. For example, the matrix multiplication $C_{ij} = A_{ik}B_{kj}$ involves three dimensions. The iteration space contains IJK points, each of which undergoes an element-wise multiplication.

(2) The order of coordinate dimensions on the right side describes the order of traversing the iteration space and accessing tensor elements during SpTA computation. Particularly, the access order of SFT must be started from the parent-fiber to the child-fiber, which should be concordant [26] with the dimension iteration order of EinSum.

(3) The correspondence between tensor indices on the left and right sides describes the relation between the output and the iteration space. Specifically, the dimensions of the output tensor are usually a subset of the iteration space. The missing dimensions are the ones that need to be reduced.

Based on these computation rules, the unified execution flow of SpTA can be abstracted as a series of iterative processing of input SFTs to construct the iteration space, apply multiplication, and reduce some dimensions to get the output SFT. The iterative processing steps contain five stages:

1. Load: access the required input fibers (nodes of SFT);

2. Intersection: for the same dimensions of multiple input SFTs, e.g., *k-crd* in $C_{ij} = A_{ik}B_{kj}$, perform intersection of fibers to find the matching coordinates and construct the iteration space together;

3. Apply: element-wise operations on each point in the constructed iteration space;

4. Merge: for the fibers in the dimension that needs to be reduced, merge its child-fibers;

5. Save: store the result SFT from the bottom to the top level.

The first three stages identify fiber elements required by element-wise computation. These stages can be seen as a join of input SFTs, which constructs the SFT of intermediate results. The last two stages are used to merge the intermediate SFT to obtain the result, which is abstracted as the merging of sub-trees. Since SFT is compatible with dense/sparse data formats and can describe multidimensional tensors, and EinSum provides a unified mathematical description, the above execution data flow abstraction can be used for general SpTA tasks with different sparsity.

3.2 FEASTA ISA Design

3.2.1 Register Definition. FEASTA ISA has two kinds of registers: **fiber register (FR)** and **configuration register (CR)**. FR has two types with different storage formats:

(1) For the last-level value fiber, FR contains the *crd* array and the *val* array, one $\{crd, val\}$ pair is a fiber element.

(2) For higher-level fibers, FR contains the *crd* array and the address information of child-fibers. The latter one consists of beginning position (*beg*) and length (*len*) of each child-fiber. One $\{crd, beg, len\}$ triple is a fiber element, and *len* indicates the number of fiber elements in child-fiber.

FR is a vectorized register storing multiple fiber elements, all in 32-bit. Therefore, computation instructions based on FR can support parallel processing of multiple fiber elements, achieving ***Design Goal 3***. CR is a scalar register holding basic SpTA information passed from the main controller: the memory address of input and output data, tensor shapes, the size of memory space allocated for intermediate results, etc. CR will not be modified by the instructions of FEASTA.

3.2.2 Basic Instruction Type. We propose seven types of FEASTA instructions for memory management and parallel processing, including `load`, `read`, `inter`, `dmr` , `acc`, `write`, and `store` (`dmr` is short for **d**ispatch, **m**ultiplication, and **r**eduction.). ISA definition is shown in Tab. 1. Each instruction is padded to 32 bits for easy access. The ISA will represent the aforementioned five-stage SFT processing. The load stage is achieved by `load` and `read` instructions. Intersections of two last-level value fibers and two other fibers are accomplished by the `dmr` and `inter` instructions, respectively. The apply and merge stages are realized through different modes of the `dmr` instruction. The `acc` instruction performs the temporal accumulation part of the merge. The save stage is fulfilled by `write` and `store` instructions.

3.2.3 *Flexibility* Design: ISA for Dense & Sparse Data. In order to achieve flexibility for SpTA tasks with different sparsity, the FEASTA ISA incorporates instruction designs

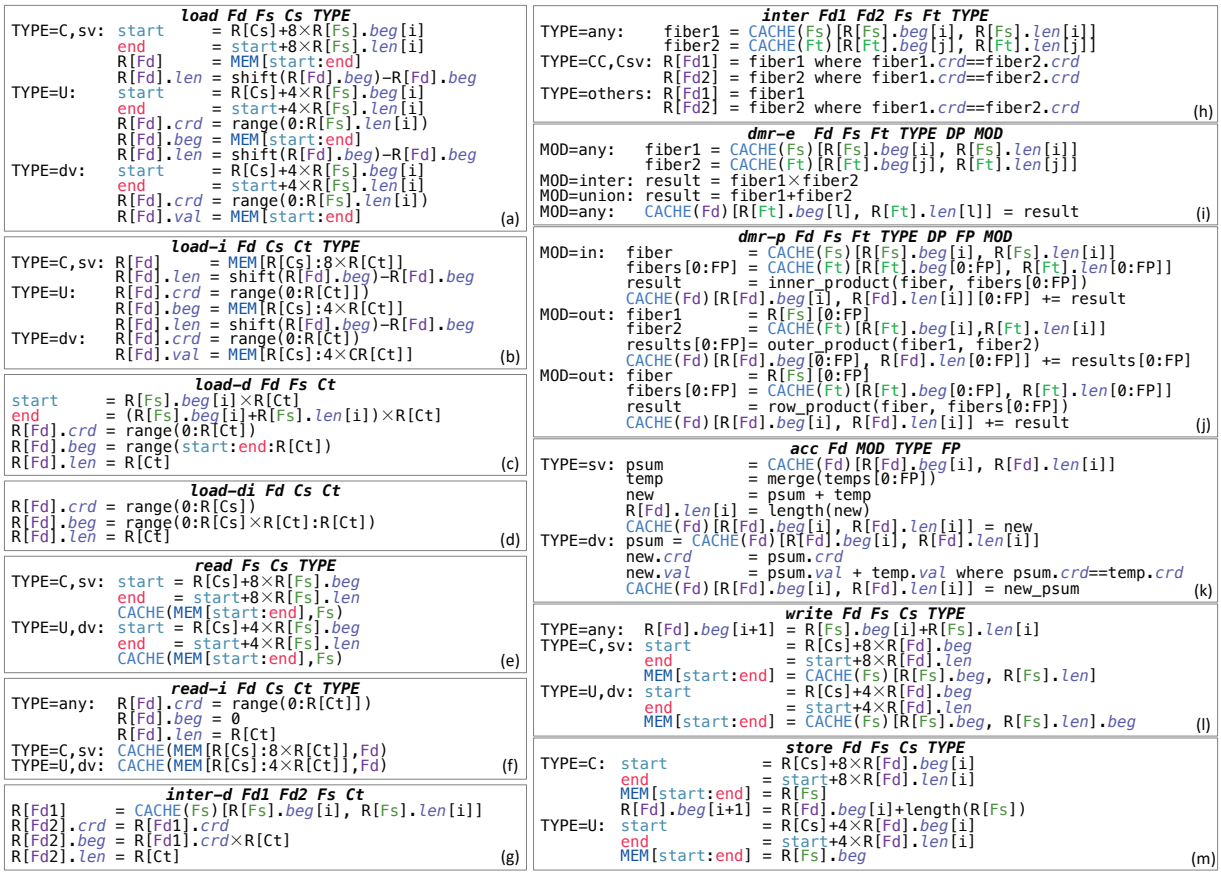


Figure 4. Semantic formula of FEASTA ISA. "R" represents registers, containing FR and CR. "range(start,end,step)" is in Python syntax and generates a list. "MEM[start,end]" means memory access from address "start" to "end". "CACHE(fiber, Fd)" means creating a space in cache labeled by "Fd" for "fiber", and "CACHE(Fd)[beg,end]" is accessing the fiber from "beg" to "end".

targeting both dense and sparse data. To this end, we first classify the fibers in SFT into five types:

(1) Dense value fiber (**dv**): value fiber refers to the last-level fiber (i.e., leaf node) in SFT. The dv fiber has a continuous *crd* array, and all values are stored sequentially. Thus, the *crd* array can be omitted in dv fibers.

(2) Sparse value fiber (**sv**): The *crd* array of sv fiber is discontinuous due to sparsity. As a result, sv fibers need to store both the *crd* array and the *val* array.

(3) Dense pointer fiber (**D**): pointer fibers are the higher-level fibers (i.e., non-leaf nodes) in SFT. The D fiber has continuous *crd* array and arithmetic *ptr* array (i.e., the lengths of child-fibers are the same, as shown in the *i*-fiber of Fig. 3(d)). In FEASTA, we only need the length of child-fiber to calculate *beg* and *len*, rather than store the *ptr* and *crd* arrays.

(4) Uncompressed sparse pointer fibers (**U**): sparse pointer fibers have continuous *crd* arrays and irregular *ptr* arrays. An example is shown in the *i*-fiber of Fig. 3(e). The continuous *crd* arrays can be omitted.

(5) Compressed sparse pointer fibers (**C**): the difference between U fibers and C fibers is that the *crd* arrays of C fibers are discontinuous and need to be stored.

To improve the utilization of the hierarchical memory, only necessary contents for each type of fiber are stored, and the instructions are designed for different types. As shown in Tab. 1, `load` and `load-i` are responsible for loading value fibers and sparse pointer fibers, while `load-d` and `load-di` load the dense pointer fibers. Besides the storage optimization, fiber types can also be used to simplify the computation. For example, if the types of two fibers are U-U, U-D, or U-dv, the intersection results are the same as the input. So, the `inter` instruction can be omitted during compilation. And the `inter-d` will not actually perform an intersection but calculate the result *beg* and *len* through the length of child-fiber, as shown in Fig. 4(g) and discussed in Sec. 4.2. Moreover, `acc` has different accumulation methods for different fiber types. For sparse fibers, the accumulation is the merge-and-sorting of fibers. `acc` configures the hardware to sort several partial sum (psum) fibers together to reduce the total merge count and psum reads, as in SpMSpM accelerators [31, 63]. For

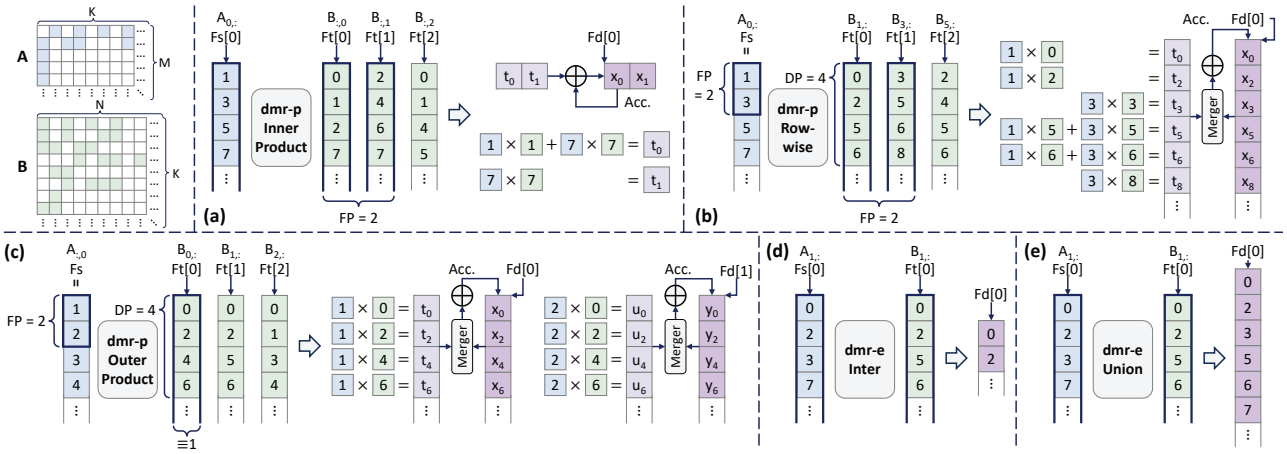


Figure 5. Dataflow for the `dmr-p` instruction to compute matrix multiplication in (a) inner-product mode, (b) row-wise mode, and (c) outer-product mode. Dataflow for the `dmr-e` instruction to compute sparse vector (d) multiplication and (e) addition. The numbers in the figure are coordinates, and values are omitted for brevity.

dense output fibers, `acc` directly accumulates psum fibers without sorting. In FEASTA ISA, the TYPE field is used to determine different execution ways and configure the hardware units, as shown in Tab. 1.

3.2.4 *Efficiency* Design: Parallelization Methods.

In order to improve efficiency, the proposed ISA supports parallelization from two aspects, i.e., parallelism within a single instruction and multi-threading parallelism among instructions. **At the single instruction level**, one instruction can process multiple fiber elements in parallel. As introduced in Sec. 3.2.1, FEASTA FRs are vectorized, which can store multiple fiber elements. The function units are also designed to operate multiple fiber elements. This is similar to vector instructions' data parallelism, and we refer to it as data parallelism (**DP**). Moreover, a single `read`, `dmr`, or `write` instruction defines operations of multiple fibers. As shown in Fig. 5(a), the `dmr-p` concurrently performs inner-products between one A fiber and a set of B fibers. We call the number of fibers in the set fiber parallelism (**FP**). The FP/DP field in the `dmr-p` instruction is to configure the parallelism. **At the multi-instruction level**, multiple same type instructions can be packed to handle more data concurrently. During runtime, the hardware modules execute these instructions in an alternating manner similar to CPU multi-threading, rather than using additional hardware resources for parallel execution or complete sequential execution. The implementation of multi-threading benefits from the fact that parallel instructions in packing do not have output data dependencies, while the input fibers can be reused in the cache. We refer to this coarse-grained multi-threading parallelism of multiple instructions as group parallelism (**GP**). When the workload of an instruction is relatively small and completed earlier, hardware modules can quickly switch to other instructions in the instruction group, avoiding resource idleness.

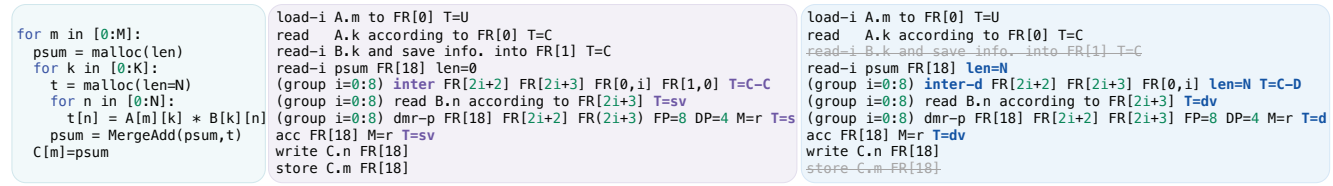
3.2.5 Detailed Instruction Semantic.

The semantic formulas of FEASTA ISA are shown in Fig. 4. The `load` instructions move the fiber from the DRAM address space to the FR address space based on the starting address and length of the fiber, as shown in Fig. 4(a-d). `load-i` gets fiber length from CR[Ct], while `load` access memory based on *beg&len* of the upper-level fiber stored in FR[Fs]. For `load-d` and `load-di`, we use the length of child-fiber to calculate D fiber's *beg* and *len*, rather than loading fibers from DRAM.

The `read` instructions are responsible for prefetching fibers from DRAM into the cache, as shown in Fig. 4(e-f). They can also be used to allocate cache space for intermediate results, e.g., psum of matrix multiplications. Storing results in an on-chip cache instead of FR has the benefit of reducing the FR occupancy and instruction count when describing parallel processing of multiple fibers. Fibers required by different instructions can be reused in the cache.

The `inter` instructions execute the intersection between two fibers, as shown in Fig. 4(g-h). Notably, the source registers (FR[Fs], FR[Ft]) do not store the input fibers but store the "pointer" (upper-level fiber) of input fibers. For example, in Fig. 6(b), A.k and B.k fibers are "read" to cache. FR[0] and FR[1] store their info. The intersector uses *beg&len* pairs in FR[0] and FR[1] to access A.k and B.k from the cache. Using cached fibers as intersection operands is because the fiber length is various, and data access rates of two fibers may vary a lot. Thus, making the intersector interact directly with the cache is more flexible. The results of `inter` are saved in FR and used by `load` and `read` of the next level SFT.

The `dmr` instructions define several modes of computation between fibers, as shown in Fig. 4(i-j). They are responsible for dispatching some last-level value fibers to multipliers followed by reduction networks, selectively performing element-wise multiplications and reducing the results. The



(a) Matrix Multiplication Pseudo Code (b) SpMSpM with FEASTA ISA ((GP)*FP*DP = (8)*8*4) (c) SpMM with FEASTA ISA ((GP)*FP*DP = (8)*8*4)

Figure 6. Example of SpMSpM and SpMM expressed using FEASTA ISA ("T" represents "TYPE", "M" represents "MOD").

different modes are shown in Fig. 5, in which the numbers are coordinates while values are omitted for brevity. The `dmr-e` instruction performs element-wise operations between two value fibers, including the union mode (Fig. 5(e)) and the intersection mode (Fig. 5(d)).The former mode supports sparse vector addition by three steps: sorting two value fibers by *crd*, dispatching sorted fibers, and adding values of matching *crd*s. The latter mode describes sparse vector element-wise multiplication, which intersects two value fibers by *crd* and dispatches them to multipliers without reduction. The `dmr-p` instruction completes three modes of tensor products: inner-product (Fig. 5(a)), row-wise product (Fig. 5(b)), and outer-product (Fig. 5(c)).

One `dmr-p` instruction is defined on one fiber of tensor A (blue in Fig. 5) and fiber(s) of tensor B (green in Fig. 5). Taking matrix multiplication A[M,K] × B[K,N] as an example, three modes correspond to three loop orders. For the M-N-K order, i.e., inner-product mode in Fig. 5(a), A-fiber is one row of A along the *k*-dim, and B-fibers are columns of B along the *k*-dim, so it's the inner-product of one A row and two B columns (FP=2). The first product is $t_1 = 1 \times 1 + 7 \times 7 + ...$ considering the matched coordinates are "1, 7". Row-wise mode means the M-K-N order; it performs the row-wise product of one A row (along *k*-dim) and matched B rows (indexed by *k*, along *n*-dim), and the multiplication results $1 \times [0\ 2\ 5\ 6]$ and $3 \times [3\ 5\ 6\ 8]$ have to be merged according to *n-crd*. Outer-product mode means the K-M-N order, A-fiber is one column of A, and B-fiber is one row of B, and it is the outer-product between [1 2] and [0 2 4 6]. *m*-dim and *n*-dim do not need to be matched, so there are eight output values.

Notably, the `dmr-p` instruction contains two parallelism dimensions: fiber parallelism indicates how many fibers are processed by one instruction, and data parallelism indicates how many elements in a fiber are processed in one cycle. They are specified by instruction fields FP/DP. As shown in Fig. 5(b), in row-wise mode, FP=2 means two elements of A-fiber (stored in FR[Fs]) will be processed, and two corresponding B-fibers will be processed (stored in the cache and represented by *beg&len* in FR[Ft]). When FP is smaller than FR size, the hardware will remember instruction execution status in "stat. buffer" (illustrated as a part of the controller in Fig. 7) and iteratively use elements in FR[Fs] as A-fiber and elements in FR[Ft] to index corresponding B-fibers.

The `acc` instruction performs psum accumulation for output fibers after `dmr-p`. For the inner-product mode, the psums are accumulated every cycle. For the row-wise-product mode and outer-product mode, the psums of different `dmr-p` instructions using the same psum cache space are accumulated.

The `write` instruction is the reverse operation of the `read` instruction, i.e., writing back fibers from the cache to DRAM. The `store` instruction is the reverse operation of the `load` instruction, i.e., writing back fiber in FR to DRAM.

3.3 FEASTA ISA Example

Fig. 6 shows SpMSpM and SpMM examples using FEASTA ISA. Both adopt the row-wise dataflow as the pseudocode in Fig. 6(a) and set different parallelism across the three independent matrix multiplication dimensions. The *m*-dim parallelism is configured through instruction group parallelism of eight (group i=0:8). The other two dimensions correspond to the fiber parallelism and data parallelism of `dmr-p` instructions. For SpMSpM in Fig. 6(b), the second instruction is `read`, A.k is read according to its *beg&len* in FR[0]. The third instruction is `read-i`, B.k is read according to CR (address and length), and its information will be stored in FR[1] for later use. For SpMM in Fig. 6(c), since the *k*-fiber of B and the *m*-fiber of C are D fibers, no `read` or `store` instructions are needed. The `inter` instruction is replaced by `inter-d`, which will calculate the *n*-fiber addresses of B based on the child-fiber length. The `dmr-p` and `acc` instructions also have TYPE fields for configuring different hardware data paths.

4 FEASTA Hardware Architecture Design

4.1 Architecture Overview

The FEASTA architecture is shown in Fig. 7. The controller buffers the instructions, manages other components, and tracks the fiber elements already processed. The memory hierarchy of FEASTA contains a register file containing FR and CR, a global on-chip cache, and an off-chip high bandwidth memory (HBM). The global cache has a unified address space to capture different data reuse patterns across operands, and the space is managed by a memory management unit (MMU), which also executes the `read` and `write` instructions. Other key components include a configurable loader and intersector, a multi-mode dispatcher, a computation array, and a

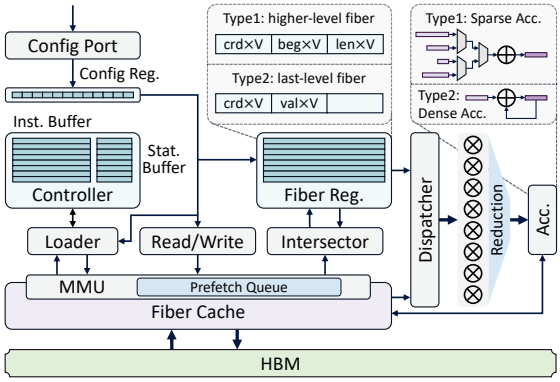


Figure 7. The overall FEASTA architecture.

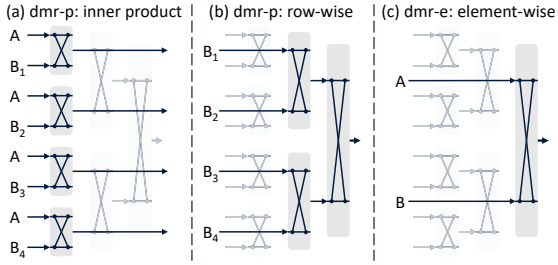


Figure 8. The dispatcher with bitonic sorter in different modes of `dmr`. (a) A and B_i are one row of a matrix and four columns of another one. (b) B_is are four rows of the right matrix. (c) A and B represent two vectors.

flexible accumulator. They are designed to process different operators with various data sparsity.

4.2 Configurable Loader and Intersector

The loader and intersector invoke different underlying hardware when processing different types of fibers. The loader reads data when executing `load` and `load-i` instructions. Depending on the TYPE field, the read fiber may contain only a *ptr* or *val* array. In this case, the loader will generate the omitted *crd* array and convert *ptr* into *beg* and *len* arrays for filling FR. While the `load-d` and `load-di` instructions do not read DRAM but compute *beg* and *len* according to child-fiber length. The data paths of the intersector also differ by fiber type. When executing `inter` instruction, the intersector uses a bitonic sorter to sort elements in two fibers according to *crd* and then compares each adjacent *crd* pair to get the intersection results. Take $[1\ 3\ 4] \cap [3\ 5]$ as an example. Sorter lines up them to $[1\ 3\ 3\ 4\ 5]$ and gets $1 < 3,\ 3 = 3,\ 3 < 4,\ 4 < 5$. Thus, 3 is the matched output. For `inter-d`, D fiber is not needed to sort. The intersection results can be directly calculated with the child fiber length and the *crd* of the other fiber. In Fig. 4(g-h), `inter` has comparisons while `inter-d` does not.

4.3 Multi-mode Dispatcher and Computation Array

The `dmr` instructions are executed with three modules: dispatcher, multiplication array, and reduction network. The reduction network adopts the FAN design from SIGMA [41] to add any number of adjacent elements. Fig. 8 explains how the dispatcher works under different modes of `dmr` instructions. For example, in `dmr-e` instructions, the dispatcher sorts two fibers with a large data parallelism and dispatches their intersection or union to downstream units for multiplication or addition. In the inner-product mode of `dmr-p` instructions, the dispatcher uses multiple parts of the bitonic sorting network to compute intersections of one *a*-fiber and multiple *b*-fibers in parallel and dispatches results. In the row-wise multiplication mode of `dmr-p`, the bitonic sorting network sorts multiple *b*-fibers, and *a*-fiber elements are selected to dispatch with the sorted *b*-fiber elements, mitigating workload imbalance. For instance, in the Spada accelerator [31], *a* and *b* fibers are directly dispatched to multipliers. When the lengths of *b*-fiber differ greatly, the overall multiplier utilization is low, increasing computation time. As introduced in Sec. 3.2.4, `dmr-p` instructions can configure different parallelism. Fig. 9 shows how the dispatcher and computation array execute at two different parallelism settings. Denoting the parallelism as (GP)×FP×DP. The parallelisms of Fig. 9(b) and (c) are (1)×2×4 and (2)×2×2, respectively. Thus, the flexible parallelism switching achieves high performance for tensors with different non-zero value distributions.

4.4 Flexible Accumulator

The accumulator adds the psum results together in three modes, as shown in Fig. 5(a)(b)(c). To support accumulations for both sparse and dense tensors, accumulators can switch between two types. For the dense type, the output of the reduction network is aligned with the psum to be accumulated, so the accumulator directly adds them. For the sparse type, merging output fibers into a psum fiber requires repeatedly reading the psum fiber and doing merge sort.

5 Discussions

5.1 Tiling and Structural Sparsity

Tiling is a common technique in tensor algebra accelerators. FEASTA can support tiling by equivalently viewing tiling as more levels in the SFT, as ExTensor [19] dose. Block sparsity is a special case of tiling, where matrices are partitioned into fixed-size blocks that are dense or zero. FEASTA can not only support block sparsity format by adding levels in SFT but also simplify computations by using D and dv fiber types.

5.2 Different Iteration Orders

The same operator can have equivalent expressions that differ in iteration orders. Matrix multiplication can be commonly implemented via three algorithms: inner-product, outer-product, and row-wise dataflow. These three orders correspond to the three modes of FEASTA's `dmr-p` and `acc`

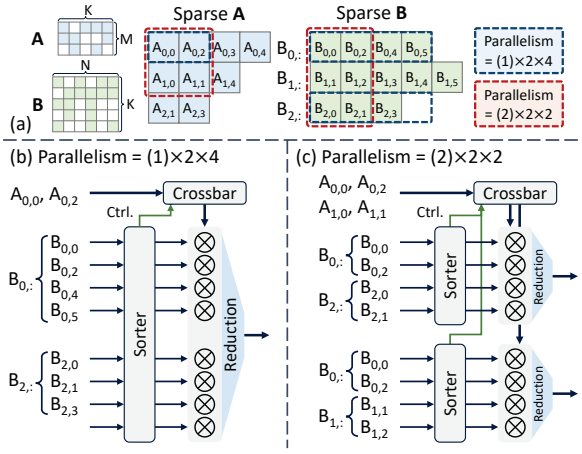


Figure 9. (a) Example of FEASTA's dispatcher and computation array to perform `dmr-p` in row-wise mode for SpMSpM, where the parallelism is (b) $(1) \times 2 \times 4$ or (c) $(2) \times 2 \times 2$.

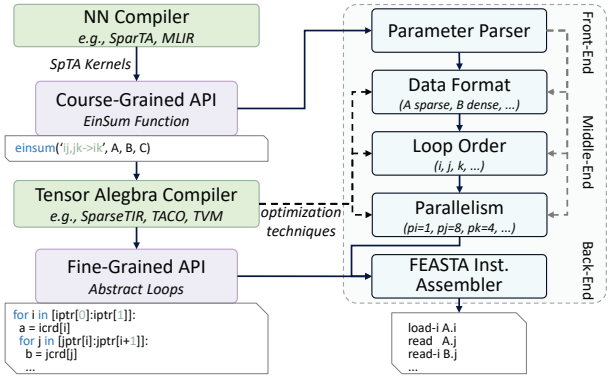


Figure 10. The compilation framework of FEASTA.

instructions, allowing FEASTA to support them all. Among SpMSpM accelerators, only Flexagon [35] supports three dataflows, but it relies on matrix partitioning to fit data entirely on-chip, and its merge unit shares the data path with the adder tree, thus has only 1-element/cycle throughput. We design the `load`, `read`, and `inter` instructions to accommodate different access orders of fibers in the three modes.

5.3 Compilation Flow

As shown in Fig. 10, FEASTA has course-grained and fine-grained user interfaces, described by the EinSum and abstract loops like [23, 26]. The former is a widely used function call in tensor libraries, reducing programmers' learning costs. The latter allows users to provide specific descriptions. Because the definition of FEASTA ISA is based on the execution flow on SFT, once SFT layout, execution order, and parallelism are obtained, translating SpTA into instructions is straightforward. We use a fixed rule compiler to generate instructions based on manually entered parameters, e.g., dimension list, data address. In practice, these parameters can be obtained by tensor compiler front-end and optimized by middle-end. Techniques in previous NN and tensor compilers like SparTA [66], SparseTIR [61], and TACO [26] will also be useful for FEASTA by connecting the interface with these compilers and doing auto optimizations to adjust data format, loop order, and parallelism. The optimized parameters and code can be sent to the FEASTA assembler back-end to generate instructions, replacing the general CPU back-end.

6 Experiments Setup

6.1 Hardware Configuration

FEASTA is configured with 64 multipliers, so the reduction network is a 64-input FAN [41]. The main component of the dispatcher is a 64-input bitonic sorter. The accumulator includes 32 tree-topology mergers; each can merge eight fibers simultaneously (same as Spada [31]). The loader contains first-in-first-out (FIFOs) for data processing and eight integer multipliers and ALUs for coordinate and address calculation. The main component of the intersector is a 32-input bitonic sorter for intersection, along with eight integer multipliers and ALUs for address calculation. The register file has 32 32-bit CRs and 32 16-element FRs. The controller contains a 128-slot instruction buffer. The cache is a 16-way 1MB cache with 64 banks for parallel data access. Data paths between cache and other components are swizzle-switch crossbars. The off-chip memory is set to 16 64-bit HBM channels with a total bandwidth of 128GB/s, and the access latency is 30ns.

6.2 Datasets and Workload

We classify the sparse tensor datasets into three sets by their applications and build different workloads with them.

Graph set. As shown in Tab. 2, we choose multiple sparse matrices from SuiteSparse [28] as previous studies [31, 63] do and add some validation datasets [56] commonly used by GNNs. We multiply sparse matrices with dense matrices with 64, 128, and 512 columns for SpMM tasks, while we multiply the sparse matrix with itself ($A \cdot A$) or its transpose ($A \cdot A^T$) for SpMSpM tasks. For SDDMM tasks, we use the sparse matrix as the sample matrix, and the dense matrices have the reduction dimension sizes of 64, 128, and 512.

CNN set. Three classic CNN models [17, 29, 46] are pruned with an open-source tool [69]. The weights and activations of layers are saved and converted by img2col, as shown in Tab. 2. We assume that there is no pruning or sparse activation to perform MM tasks between dense weights and activations. We use pruned weight matrices and dense activation matrices to perform SpMM. The zero values caused by the ReLU function [37] are filtered out for SpMSpM tasks.

Transformer set. The attention map of the Transformer can also be pruned [13], and its calculation can be seen as SDDMM tasks. And the following multiplications are thus

Table 2. Sparse matrices in Graph, CNN, and Transformer.

Set	Data	Density	Data	Density
Graph Set	web-Google	6.08e-06	citeseer	8.22e-04
	Hardesty2	1.42e-05	cora	1.44e-03
	nell	5.82e-05	poisson3Da	1.93e-03
	pubmed	2.28e-04	Computers	2.60e-03
	email-Enron	2.73e-04	lpi_forest6	2.85e-02
	ca-CondMat	3.35e-04	cari	3.18e-01
CNN Set	AlexNet_3_W	6.38e-01	AlexNet_3_A	6.17e-01
	AlexNet_8_W	5.94e-01	AlexNet_8_A	1.73e-01
	ResNet18_3_W	2.31e-01	ResNet18_3_A	1.72e-01
	ResNet18_4_W	6.36e-02	ResNet18_4_A	1.21e-01
	Vgg16_27_W	3.95e-01	Vgg16_27_A	2.44e-01
	Vgg16_40_W	5.59e-01	Vgg16_40_A	1.51e-01
TF Set	bert_1024	5.00e-02	opt_1024	1.87e-02

SpMM tasks. As shown in Tab.2, we derive matrices from BERT [11] and OPT-6.7B [64] and build SpMM and SDDMM tasks with dense matrices with 512 or 1024 columns.

6.3 Baselines

We compare FEASTA against state-of-the-art accelerators optimized for different kernels, a CPU extension, and GPU.

Tensor Core (TC): The acceleration architecture for dense matrix multiplication is proposed by NVIDIA on the Volta architecture GPUs. We take its architecture as an accelerator.

Tensaurus [49]: An accelerator targeting sparse-dense operators like SpMM and SpMTTKRP.

Spada [31]: The state-of-the-art SpMSpM accelerator using row-wise dataflow. It can dynamically adjust the row and column parallelism of one matrix.

SparseCore [42]: A CPU instruction set extension of key-value stream data types, the merge instructions can be used to accelerate merging in row-wise SpMSpM.

GPU: We run SpTA operators with NVIDIA's CuSparse library [38] on a V100S GPU.

6.4 Evaluation Methodology

In order to compare the hardware cost and performance, we set the number of multipliers for all architectures to 64. And the other parts are adjusted accordingly. TC and Tensaurus are set to have 512KB scratchpad memory. Spada and FEASTA both have a 1MB cache. In all experiments, 32-bit floating-point data is used for values, and 32-bit integer is used for coordinates, which is usually sufficient in machine learning tasks. The off-chip memory of each architecture is uniformly set as 128GB/s bandwidth with a latency of 30ns. When evaluating SpMM and SpMSpM on the TC architecture, all sparse matrices are in a dense format that stores zero values. When evaluating Tensaurus' MM task and Spada's SpMM and MM tasks, the dense matrix is encoded as a sparse matrix of 0% sparsity (storing coordinates).

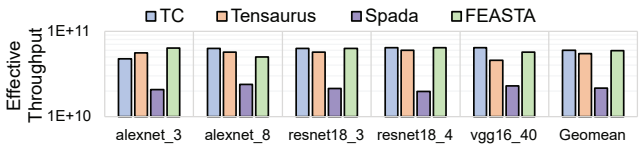


Figure 11. Performance comparison on MM tasks.

We use Spada's open-source cycle-accurate simulator to model its performance. Based on the simulation API of on-chip memory and HBM in the Spada simulator, we implement the data path and control logic for TC, Tensaurus, and FEASTA. Therefore, we ensure that the performance of each architecture is evaluated with the same memory interface. TC and Tensaurus have fixed dataflow, and we implement their control logic as a finite state machine that iterates tiles on dimensions. For FEASTA, we use the same simulation principle as Spada. Timing models of function units are implemented, and the FEASTA simulator controls the order of data transmission among units depending on the instructions, while Spada uses a fixed transmission manner.

We implement modules of FEASTA in Verilog, including controller, loader, intersector, dispatcher, multiplier, accumulator, and reduction network. The key components of baselines are also reproduced. All modules are synthesized using Synopsys Design Compiler with TSMC 28nm technology at 1GHz frequency to get the area and power. CACTI [36] is used to evaluate the area and power for on-chip memory, including register files, scratchpad memory, and cache.

7 Evaluation

7.1 Kernel-Level Performance

We evaluate accelerators by effective throughput, which is defined as the number of non-zero multiplications in a task divided by latency. Fig. 11 shows the results of MM. We can see that TC designed for MM has the best performance. FEASTA achieves 97% of the TC performance, but Spada can not. This is because Spada's fixed data path has to treat dense matrices as 0% sparsity and introduce overheads. FEASTA's instruction-driven data path can not only process sparse data but also calculate addresses of dense fibers and load values without coordinates. This is exactly what TC does.

Fig. 12 shows the performance of SpMM. The throughput of FEASTA is 1.26× Spada on graph datasets. We can see that TC processes zeros in the sparse matrix, drastically reducing the effective throughput on graph datasets. But it is good on CNN datasets because the sparsity of CNN is small. FEASTA is configured to run SpMM in this experiment so it can achieve performance similar to Tensaurus.

For SpMSpM, TC and Tensaurus have low effective throughputs on graph datasets, as shown in Fig. 13. FEASTA's geomean performance is 1.47× higher than Spada's because of workload balance. The parallelism of FEASTA and Spada

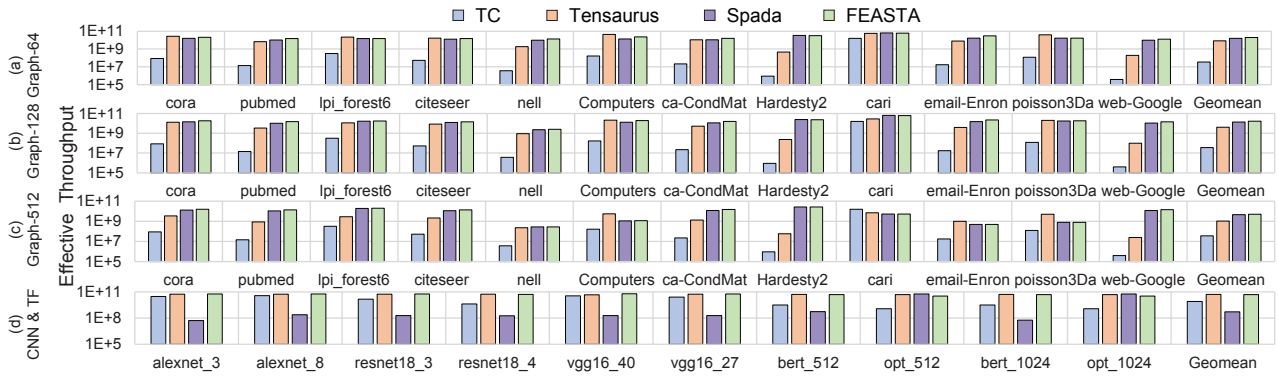


Figure 12. Performance comparison on SpMM tasks.

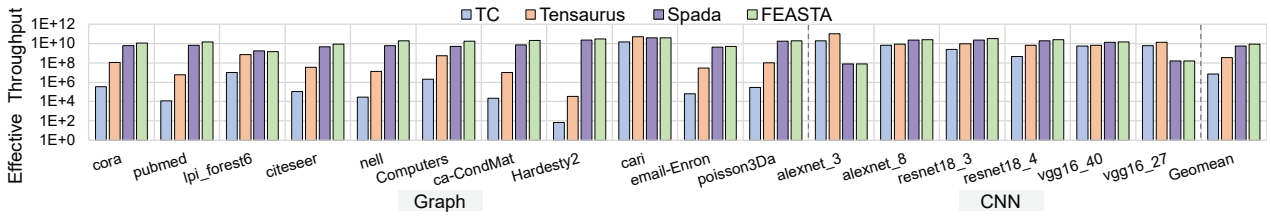


Figure 13. Performance comparison on SpMSpM tasks.

is both 8x8. Spada dispatches workload according to row ID, which is vulnerable to unbalanced element distribution. Because it calculates eight A-rows with fixed eight PEs, the latency is limited by the densest row. In contrast, FEASTA processes one A-row per instruction and processes eight rows concurrently by group parallelism, as described in Sec. 3.2.4. All workload is processed by all PEs, reducing the impact of unbalanced workload. Also, FEASTA's dispatching before multiplication mechanism helps balance B matrix load, as discussed in Sec. 4.3. The largest matrix is Hardesty, with more than 900k nodes. The density varies from 0.0006% to 0.3%, as shown in Tab. 2. Fig. 13 shows that FEASTA outperforms Spada on most matrices regardless of size and sparsity. They are worse than Tensaurus on two CNN tasks because the sparsity is low. In fact, FESTA can be configured to run them as SpMM and achieve performance similar to Tensaurus.

No baseline can execute SDDMM as a sparse kernel, so we evaluated the performance of TC (suitable for MM) to execute SDDMM in a dense way. The comparison is shown in Fig. 15 and Fig. 14. Performing SDDMM in a sparse way with FEASTA achieves 20.65× geomean throughput than performing it in a dense way with TC on TF datasets, and up to 422.16× geomean throughput on graph datasets.

7.2 Model-Level Performance

Besides-kernel level tasks, the end-to-end performance of NN models is also evaluated. The GNN models are GCNs on Cora, Citeseer, and Pubmed. The CNN models are VGG16 and ResNet50, and the LLM model is LLaMA2-7B. We extract all

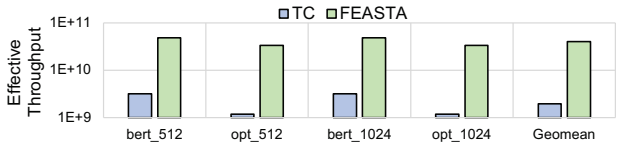


Figure 14. Performance of SDDMM tasks in transformers.

matrix operators from the network, run them one by one in simulators, and add up the results as the end-to-end latency. The original CNN and LLM have no sparsity. We prune the weights of GNN and CNN [16], with 60%-90% unstructured sparsity. The ReLU function sparsifies the activation, and the sparsity is 44%-78%. For LLM, the average sparsity of pruned weight [15], activation [34], and attention is 54%, 61%, and 81%. Since pruning is optional for NN models, both sparse and dense kernels are tested, and the best performance for each accelerator is selected as the final result.

As shown in Fig. 16, FEASTA achieves the best geomean performance among models. Because FEASTA is flexible and can choose sparse or dense processing manners for different kernels, it can achieve better performance in mixed tasks containing different types of SpTA and sparsity compared to other architectures with only one fixed processing mode.

7.3 Area and Power Evaluation

Tab. 3 shows the area of FEASTA. Similar to other kernel accelerators [31, 35, 63], the main area of FEASTA comes from the cache. In this regard, we follow other accelerators and

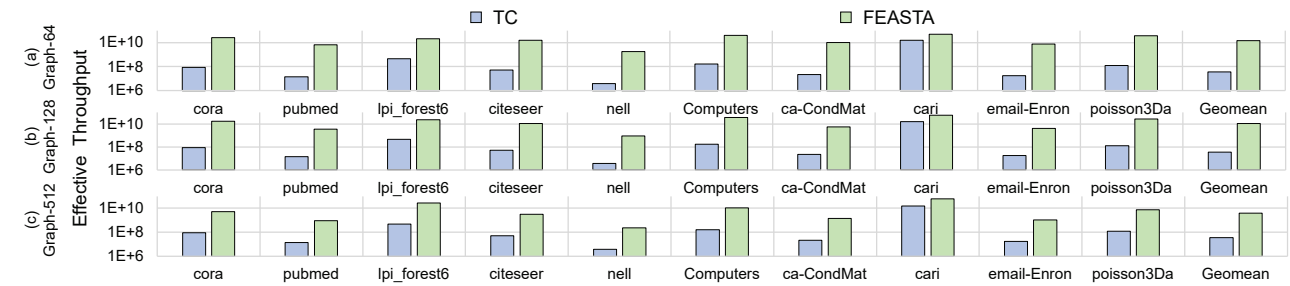


Figure 15. Performance comparison on SDDMM tasks on graph datasets.

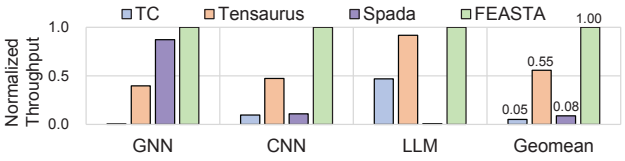


Figure 16. Model-level performance comparison.

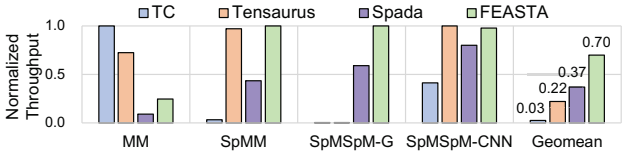


Figure 17. ISO-area performance comparison.

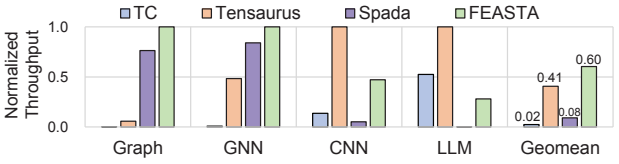


Figure 18. ISO-area comparison on model-level tasks.

set a 1MB cache to alleviate memory access bottlenecks in highly sparse tasks and support flexible concurrent memory access. In addition, to support different dataflow of various SpTA, FEASTA introduces some overhead of register files for the flexible architecture. The dispatcher and crossbar are more complicated than single dataflow architecture.

The area of TC, Tensaurus, Spada, and FEASTA is $0.94mm^2$, $0.96mm^2$, $4.07mm^2$, and $4.39mm^2$, while the power of them is 3.21W, 3.96W, 10.44W, and 12.51W, respectively. It can be seen that architectures designed for operators with more sparsity have larger areas and power. For Spada and FEASTA, which can support SpMSpM, their area is not significantly different. Due to the on-chip cache, the area of Spada and FEASTA is much larger than TC and Tensaurus. But from Fig. 13, we can see from the geomean results that Spada and FEASTA supporting sparse matrix processing can improve throughput hundreds of times that of TC. So, designing an accelerator for SpTA is very necessary and meaningful.

Table 3. The area breakdown of FEASTA.

Module	Area (mm^2)	Module	Area (mm^2)
Controller & RF	0.08	Crossbar	0.17
Loader	0.14	Intersector	0.06
Dispatcher	0.15	Multipliers	0.14
Reduction	0.38	Accumulator	0.08
Fiber Cache	3.19	**Total**	**4.39**

Together with the geomean performance results shown in Fig. 1, we can give the area efficiency and power efficiency analysis. The area efficiency ratio of TC, Tensaurus, Spada, and FEASTA is 1 : 5.9 : 5.7 : 10.6, and the power efficiency ratio is 1 : 4.9 : 7.5 : 12.7. Both area and energy efficiency of FEASTA outperforms those of baselines on heterogeneous workload. To support general SpTA, accelerators have to introduce more units of decoding, sorting, and cache, so Spada and FEASTA have larger areas than TC and have lower area efficiency on only MM tasks. FEASTA's goal is not to outperform TC on MM but to solve how to design hardware for various tensor algebra workloads with sparse and dense mixtures. SpTA with different sparsity is common in the real world. TC is simple and efficient for MM, but suffers from low utilization on sparse workload. FEASTA and Spada can handle it in a sparse manner. FEASTA is more flexible than Spada in processing dense tensors in a simplified way and supports more operators like SDDMM.

7.4 ISO-area Evaluation

We also perform an equal-area comparison besides equal-multiplier experiments. We increase the number of computing units for TC and Tensaurus by eight times (two times on the output row dimension and four times on the output column dimension) so that their area is similar to that of Spada and FEASTA. The parallelism of on-chip buffers also changes, but the total amount remains less than 1MB. The same methodology as in Sec 6.4 is used to evaluate the performance and hardware cost. The area of TC, Tensaurus, Spada, and FEASTA is $4.23mm^2$, $4.55mm^2$, $4.07mm^2$, and $4.39mm^2$, while the power of them is 18.61W, 23.71W, 10.44W, and 12.51W. The performance results are shown in Fig 17. It can

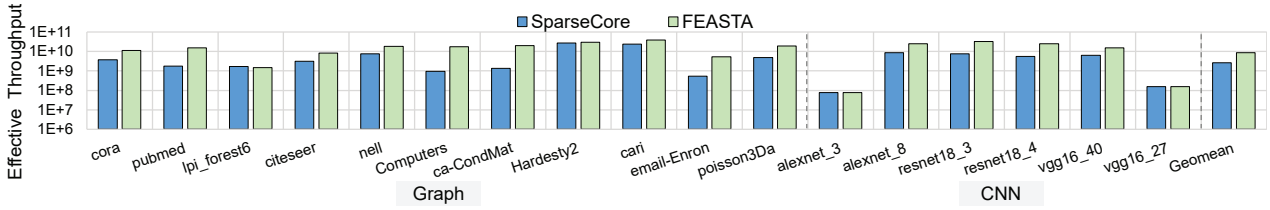


Figure 19. Performance comparison with SparseCore on SpMSpM tasks.

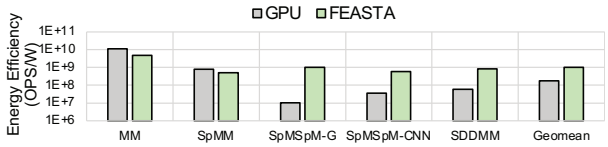


Figure 20. Energy efficiency of FEASTA and GPU.

be seen that FEASTA performs worse than TC on MM tasks because TC is simple and can achieve larger parallelism in the same area. But because TC cannot handle other SpTA kernels sparsely, FEASTA achieves better performance on other tasks and, therefore, has higher geomean performance. The model-level performance under ISO-area condition is also evaluated, and the results are shown in Fig. 18, where FEASTA achieved the best geomean performance. Considering power efficiency, FEASTA exceeds other architectures by more than 2.8×. Tensaurus performs best on LLM due to its larger parallelism and the lower sparsity in LLM.

7.5 Comparison with SparseCore

This section compares with SparseCore, another flexible design. Notably, as a CPU ISA extension for key-value streams, SparseCore inherits flexibility from CPU and only accelerates sparse psum accumulation for SpTA using the merging instruction and unit. Thus, we simulate a row-wise SpMSpM operator and implement all vector merging with a 64-parallel sorting unit. Fig. 19 shows the result, where SparseCore only merges two vectors, but FEASTA performs concurrent row-wise products across fibers, leading to 3.19× performance.

7.6 Comparison with GPU

We measure the latency of kernels running on GPU to get its effective throughput. GPU has higher throughput on MM and SpMM since it has much more computation resources but with larger area and power. Considering power efficiency, FEASTA is 5.40× power efficient than GPU among kernels and up to 97.44× on SpMSpM kernels, as shown in Fig. 20.

8 Related Work

8.1 Sparse Tensor Algebra Modeling

TACO [25] proposes the theory of SpTA compiler. Capstan [44] connects the SpTA to a low-level map-reduce-based programming model [27]. Sparseloop [60] categorizes sparse acceleration techniques. SparTA [66] designs a tensor abstraction for pruning neural networks. SAM [23] proposes some streaming dataflow blocks to generate SpTA accelerators. SparseTIR [61] proposes formats and transformation abstractions for sparse deep learning acceleration.

8.2 Sparse Kernel Accelerators

There are many accelerators designed for specific sparse kernels. [45] was the first to research multi-way merge parallelization methods for sparse matrix-vector multiplication (SpMV). CoSparse [14] and SparseAdapt [39] proposed a reconfigurable framework to accelerate SpMV and SpMM, while HiSparse [12] and Sextans [47] created streaming accelerators that utilize the HBM on FPGA. SIGMA [41] and Sparse-TPU [18] adopt the inner product dataflow to support SpMSpM kernels, while OuterSPACE [40], SpArch [65], and Spaghetti [22] explore the outer product dataflow. SOTA SpMSpM accelerators [31, 48, 63] use row-wise dataflow, which combines the advantages of the former two.

9 Conclusion

This paper proposes FEASTA, a flexible and efficient accelerator for SpTA. There are three main techniques: a unified modeling for SpTA execution flow, a flexible SpTA ISA with an efficient parallelization mechanism, and an instruction-driven configurable architecture. Evaluations on various tasks show that FEASTA achieves 5.66× geomean power efficiency improvements compared to GPU. FEASTA delivers 1.47× and 3.19× higher performance on sparse matrix multiplication kernels compared to SOTA sparse matrix accelerator and CPU extension, and achieves 1.69-12.70× energy efficiency over existing architectures across diverse kernels.

Acknowledgments

This work was supported by National Natural Science Foundation of China (No. 62325405, 62104128, U19B2019, U21B2031, 61832007, 62204164), and Beijing National Research Center for Information Science and Technology (BNRist). We thank Tianyu Fu, Lidong Guo from Tsinghua University, and Jun Liu from Shanghai Jiao Tong University for their valuable discussions and assistance in the experiment.

References

[1] Sriram Aananthakrishnan, Nesreen K. Ahmed, Vincent Cave, Marcelo Cintra, Yigit Demir, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, Ivan Ganev, Wim Heirman, Hans-Christian Hoppe, Jason Howard, Ibrahim Hur, MidhunChandra Kodiyath, Samkit Jain, Daniel S. Klowden, Marek M. Landowski, Laurent Montigny, Ankit More, Przemyslaw Ossowski, Robert Pawlowski, Nick Pepperling, Fabrizio Petrini, Mariusz Sikora, Balasubramanian Seshasayee, Shaden Smith, Sebastian Szkoda, Sanjaya Tayal, Jesmin Jahan Tithi, Yves Vandriessche, and Izajasz P. Wrosz. Piuma: programmable integrated unified memory architecture. arXiv preprint arXiv:2010.06277, 2020.

[2] Krister Åhlander. Einstein summation for multi-dimensional arrays. 2000.

[3] Maximiliana Behnke and Kenneth Heafield. Losing heads in the lottery: Pruning transformer attention in neural machine translation. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 2664–2674, 2020.

[4] Vivek Bharadwaj, Aydın Buluç, and James Demmel. Distributed-memory sparse kernels for machine learning. 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 47–58, 2022.

[5] Haodong Bian, Gangsheng Li, Linbing Liu, Dongqiang Huang, Runting Dong, and Jianqiang Huang. Research on accelerating the performance of spmv based on avx2 instruction set. Proceedings of the 2020 4th High Performance Computing and Cluster Technologies Conference & 2020 3rd International Conference on Big Data and Artificial Intelligence, 2020.

[6] Siheng Chen, Baoan Liu, Chen Feng, Carlos Vallespi-Gonzalez, and Carl Wellington. 3d point cloud processing and learning for autonomous driving: Impacting map creation, localization, and perception. IEEE Signal Processing Magazine, 38(1):68–86, 2020.

[7] Jack Choquette, Olivier Giroux, and Denis Foley. Volta: Performance and programmability. Ieee Micro, 38(2):42–52, 2018.

[8] Christopher Bongsoo Choy, JunYoung Gwak, and Silvio Savarese. 4d spatio-temporal convnets: Minkowski convolutional neural networks. 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pages 3070–3079, 2019.

[9] William J Dally, Stephen W Keckler, and David B Kirk. Evolution of the graphics processing unit (gpu). IEEE Micro, 41(6):42–51, 2021.

[10] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights. Proceedings of the IEEE, 109:1706–1752, 2020.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. ArXiv, abs/1810.04805, 2019.

[12] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv. Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2022.

[13] Aosong Feng, Irene Li, Yuang Jiang, and Rex Ying. Diffuser: Efficient transformers with multi-hop attention diffusion for long sequences. arXiv preprint arXiv:2210.11794, 2022.

[14] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong hyeon Park, John Magnus Morton, Trevor N. Mudge, Murray Cole, Michael F. P. O'Boyle, Chaitali Chakrabarti, and Ronald G. Dreslinski. Cosparse: A software and hardware reconfigurable spmv framework for graph analytics. 2021 58th ACM/IEEE Design Automation Conference (DAC), pages 949–954, 2021.

[15] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In International Conference on Machine Learning, pages 10323–10337. PMLR, 2023.

[16] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149, 2015.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[18] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong hyeon Park, Austin Rovinski, Haojie Ye, Kuan-Yu Chen, Ronald G. Dreslinski, and Trevor N. Mudge. Sparse-tpu: adapting systolic arrays for sparse matrices. Proceedings of the 34th ACM International Conference on Supercomputing, 2020.

[19] Kartik Hegde, Hadi Asghari Moghaddam, Michael Pellauer, Neal Clayton Crago, Aamer Jaleel, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. Extensor: An accelerator for sparse tensor algebra. Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019.

[20] Eric Hein, Tom Conte, Jeffrey Young, Srinivas Eswar, Jiajia Li, Patrick Lavin, Richard Vuduc, and Jason Riedy. An initial characterization of the emu chick. In 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 579–588. IEEE, 2018.

[21] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. J. Mach. Learn. Res., 22:241:1–241:124, 2021.

[22] Reza Hojabr, Alireza Sedaghati, Amir Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. Spaghetti: Streaming accelerators for highly sparse gemm on fpgas. 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pages 84–96, 2021.

[23] Olivia Hsu, Maxwell Strange, Jaeyeon Won, Ritvik Sharma, Kunle Olukotun, Joel S. Emer, Mark Horowitz, and Fredrik Kjolstad. The sparse abstract machine. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, 2022.

[24] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Gespmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–12, 2020.

[25] Fredrik Kjolstad. Sparse Tensor Algebra Compilation. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb 2020.

[26] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. Proc. ACM Program. Lang., 1(OOPSLA):77:1–77:29, October 2017.

[27] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 296–311, 2018.

[28] Scott P. Kolodziej, Mohsen Mahmoudi Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandström. The suitesparse matrix collection website interface. J. Open Source Softw., 4:1244, 2019.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60:84 – 90, 2012.

[30] Ying Li, Lingfei Ma, Zilong Zhong, Fei Liu, Michael A Chapman, Dongpu Cao, and Jonathan Li. Deep learning for lidar point clouds in autonomous driving: A review. IEEE Transactions on Neural Networks and Learning Systems, 32(8):3412–3432, 2020.

[31] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. Spada: Accelerating sparse matrix multiplication with adaptive dataflow. Proceedings of the 28th ACM

International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2023.

[32] Liu Liu, Zheng Qu, Zhaodong Chen, Fengbin Tu, Yufei Ding, and Yuan Xie. Dynamic sparse attention for scalable transformer acceleration. IEEE Transactions on Computers, 71(12):3165–3178, 2022.

[33] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021.

[34] Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. Relu strikes back: Exploiting activation sparsity in large language models. arXiv preprint arXiv:2310.04564, 2023.

[35] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L Abellán, Manuel E Acacio, and Tushar Krishna. Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, pages 252–265, 2023.

[36] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Cacti 6 . 0 : A tool to understand large caches. 2007.

[37] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In International Conference on Machine Learning, 2010.

[38] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. Cusparse library. In GPU Technology Conference, 2010.

[39] Subhankar Pal, Aporva Amarnath, Siying Feng, Michael F. P. O'Boyle, Ronald G. Dreslinski, and Christophe Dubach. Sparseadapt: Runtime control for sparse linear algebra on a reconfigurable accelerator. MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021.

[40] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 724–736. IEEE, 2018.

[41] Eric Qin, A. Samajdar, Hyoukjun Kwon, V Ramana Pavan Nadella, Sudarshan M. Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 58–70, 2020.

[42] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. Sparsecore: stream isa and processor specialization for sparse computation. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pages 186–199, 2022.

[43] Hongyu Ren, Hanjun Dai, Zihang Dai, Mengjiao Yang, Jure Leskovec, Dale Schuurmans, and Bo Dai. Combiner: Full attention transformer with sparse computation cost. In Neural Information Processing Systems, 2021.

[44] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. Capstan: A vector rda for sparsity. In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, pages 1022–1035, 2021.

[45] Fazle Sadi, Joseph Sweeney, Tze Meng Low, James C. Hoe, Lawrence T. Pileggi, and Franz Franchetti. Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization. Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019.

[46] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[47] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young kyu Choi, Jason Lau, and Jason Cong. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2021.

[48] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 766–780. IEEE, 2020.

[49] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 689–702. IEEE, 2020.

[50] Vivienne Sze, Yu hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks. Synthesis Lectures on Computer Architecture, 2020.

[51] Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. Advances in neural information processing systems, 33:6377–6389, 2020.

[52] Erik H Thiede, Wenda Zhou, and Risi Kondor. Graph neural networks for biochemistry that incorporate substructure. Biophysical Journal, 121(3):531a, 2022.

[53] Bangyan Wang, Lei Deng, Fei Sun, Guohao Dai, L. Liu, Yu Wang, and Yuan Xie. A one-for-all and o(v log(v))-cost solution for parallel merge style operations on sorted key-value arrays. Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022.

[54] Cheng Wang, Ming Cheng, Ferdous Sohel, Bennamoun, and Jonathan Li. Normalnet: A voxel-based cnn for 3d object classification and retrieval. Neurocomputing, 323:139–147, 2019.

[55] Hanrui Wang, Kuan Wang, Jiacheng Yang, Linxiao Shen, Nan Sun, Hae-Seung Lee, and Song Han. Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In 2020 57th ACM/IEEE Design Automation Conference (DAC), pages 1–6. IEEE, 2020.

[56] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yujie Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Haotong Zhang, Haibin Lin, Junbo Jake Zhao, Jinyang Li, Alex Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. ArXiv, abs/1909.01315, 2019.

[57] Xiang Wang, Tinglin Huang, Dingxian Wang, Yancheng Yuan, Zhenguang Liu, Xiangnan He, and Tat-Seng Chua. Learning intents behind interactions with knowledge graph for recommendation. In Proceedings of the Web Conference 2021, pages 878–887, 2021.

[58] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An adaptive and efficient runtime system for gnn acceleration on gpus. In USENIX Symposium on Operating Systems Design and Implementation, 2020.

[59] Felix Wu, Tianyi Zhang, Amauri H. de Souza, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks. In International Conference on Machine Learning, 2019.

[60] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S Emer. Sparseloop: An analytical approach to sparse tensor accelerator modeling. In 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1377–1395. IEEE, 2022.

[61] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. arXiv preprint arXiv:2207.04606, 2022.

[62] Zhe Yuan, Yongpan Liu, Jinshan Yue, Yixiong Yang, Jingyu Wang, Xiaoyu Feng, Jian Zhao, Xueqing Li, and Huazhong Yang. Sticker:

An energy-efficient multi-sparsity compatible accelerator for convolutional neural networks in 65-nm cmos. IEEE Journal of Solid-State Circuits, 55:465–477, 2020.

[63] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel 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, pages 687–701, 2021.

[64] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. ArXiv, abs/2205.01068, 2022.

[65] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 261–274. IEEE, 2020.

[66] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. {SparTA}:{Deep-Learning} model sparsity via {Tensor-with-Sparsity-Attribute}. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 213–232, 2022.

[67] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. AI open, 1:57–81, 2020.

[68] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 359–371, 2019.

[69] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. Neural network distiller: A python package for dnn compression research. ArXiv, abs/1910.12232, 2019.

A Original data

Four tables are the original data of Fig. 11 (Unit: FLOPS).

Graph-64	cora	pubmed	lpi_forest6	citeseer	nell	Computers	ca-CondMat	Hardesty2	cari	email-Enron	poisson3Da	web-Google
TC	8.72E+07	1.44E+07	3.13E+08	5.26E+07	3.72E+06	1.65E+08	2.22E+07	9.11E+05	1.57E+10	1.74E+07	1.23E+08	3.89E+05
Tensaurus	2.67E+10	6.77E+09	2.23E+10	1.66E+10	1.82E+09	4.30E+10	1.06E+10	4.62E+08	5.63E+10	7.92E+09	4.00E+10	1.98E+08
Spada	1.61E+10	1.04E+10	1.49E+10	1.30E+10	9.49E+09	1.32E+10	1.08E+10	3.28E+10	6.37E+10	1.68E+10	1.65E+10	9.69E+09
FEASTA	2.03E+10	1.50E+10	1.51E+10	1.48E+10	1.33E+10	2.31E+10	1.54E+10	3.09E+10	5.99E+10	3.08E+10	1.70E+10	1.29E+10

Graph-128	cora	pubmed	lpi_forest6	citeseer	nell	Computers	ca-CondMat	Hardesty2	cari	email-Enron	poisson3Da	web-Google
TC	8.72E+07	1.44E+07	3.17E+08	5.26E+07	3.72E+06	1.65E+08	2.22E+07	9.11E+05	1.57E+10	1.74E+07	1.23E+08	3.89E+05
Tensaurus	1.34E+10	3.38E+09	1.12E+10	8.29E+09	9.08E+08	2.15E+10	5.28E+09	2.31E+08	2.82E+10	3.96E+09	2.00E+10	9.92E+07
Spada	1.42E+10	1.04E+10	1.69E+10	1.22E+10	2.29E+09	1.30E+10	1.13E+10	2.42E+10	6.46E+10	1.48E+10	1.71E+10	1.07E+10
FEASTA	1.80E+10	1.50E+10	1.75E+10	1.43E+10	2.47E+09	1.93E+10	1.60E+10	2.37E+10	6.17E+10	2.25E+10	1.80E+10	1.44E+10

Graph-512	cora	pubmed	lpi_forest6	citeseer	nell	Computers	ca-CondMat	Hardesty2	cari	email-Enron	poisson3Da	web-Google
TC	8.72E+07	1.44E+07	3.19E+08	5.26E+07	3.72E+06	1.65E+08	2.22E+07	9.11E+05	1.57E+10	1.74E+07	1.23E+08	3.89E+05
Tensaurus	3.34E+09	8.46E+08	2.79E+09	2.07E+09	2.27E+08	5.37E+09	1.32E+09	5.78E+07	7.04E+09	9.90E+08	5.00E+09	2.48E+07
Spada	1.25E+10	1.05E+10	1.89E+10	1.11E+10	2.78E+08	1.11E+09	1.17E+10	2.56E+10	5.10E+09	4.86E+08	7.65E+08	1.15E+10
FEASTA	1.53E+10	1.36E+10	1.99E+10	1.32E+10	2.80E+08	1.13E+09	1.48E+10	2.54E+10	5.09E+09	4.90E+08	7.67E+08	1.41E+10

CNN&TF	alexnet_3	alexnet_8	resnet_3	resnet_4	vgg_40	512		1024	
						bert	opt	bert	opt
TC	3.05E+10	3.73E+10	1.46E+10	4.07E+09	3.58E+10	3.20E+09	1.20E+09	3.20E+09	1.20E+09
Tensaurus	5.43E+10	5.34E+10	5.33E+10	5.47E+10	4.52E+10	5.23E+10	4.93E+10	5.23E+10	4.93E+10
Spada	5.34E+07	2.47E+08	1.91E+08	1.82E+08	1.90E+08	5.44E+08	5.66E+10	5.94E+07	5.65E+10
FEASTA	5.87E+10	5.85E+10	5.74E+10	5.12E+10	5.94E+10	4.89E+10	3.39E+10	4.89E+10	3.39E+10