# Cerberus: Triple Mode Acceleration of Sparse Matrix and Vector Multiplication

SOOJIN HWANG, DAEHYEON BAEK, JONGSE PARK, and JAEHYUK HUH, KAIST, Yuseong-gu, Republic of Korea

The multiplication of sparse matrix and vector (SpMV) is one of the most widely used kernels in high-performance computing as well as machine learning acceleration for sparse neural networks. The design space of SpMV accelerators has two axes: algorithm and matrix representation. There have been two widely used algorithms and data representations. Two algorithms, scalar multiplication and dot product, can be combined with two sparse data representations, compressed sparse and bitmap formats for the matrix and vector. Although the prior accelerators adopted one of the possible designs, it is yet to be investigated which design is the best one across different hardware resources and workload characteristics. This paper first investigates the impact of design choices with respect to the algorithm and data representation. Our evaluation shows that no single design always outperforms the others across different workloads, but the two best designs (i.e. compressed sparse format and bitmap format with dot product) have complementary performance with trade-offs incurred by the matrix characteristics. Based on the analysis, this study proposes Cerberus, a triple-mode accelerator supporting two sparse operation modes in addition to the base dense mode. To allow such multi-mode operation, it proposes a prediction model based on matrix characteristics under a given hardware configuration, which statically selects the best mode for a given sparse matrix with its dimension and density information. Our experimental results show that Cerberus provides 12.1× performance improvements from a dense-only accelerator, and 1.5× improvements from a fixed best SpMV design.

CCS Concepts: • **Hardware** → **Hardware accelerators**.

Additional Key Words and Phrases: Sparse Matrix-Vector Multiplication (SpMV), Accelerator

## 1 INTRODUCTION

The multiplication of sparse matrix and vector (SpMV) has been one of the key computations in high-performance computing (HPC), including recommendation systems [41], linear programming [43], graph analytics [7], and big-data analytics [8]. In addition to such HPC applications, the computation of sparse neural networks employs SpMV as the basic building block in neural network accelerators [15, 17, 52]. Recent transformer accelerators also use matrix-vector multiplicator as the primary computation engine [27].

Due to the importance of SpMV operation, there have been several prior accelerator studies that leveraged the unique value property and explored custom hardware acceleration [17, 18, 52]. While these accelerators have their own unique designs, there are two major architectural design choices: algorithm and matrix format. Although software kernels for GPU/CPU have considered variants of such algorithms and formats, the advent of accelerators

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) (IITP2017-0-00466 SW StarLab and IITP2021-0-01817 Development of Next-Generation Computing Techniques for Hyper-Composable Datacenters), funded by the Ministry of Science and ICT, Korea.

Authors' address: Soojin Hwang, sjhwang@casys.kaist.ac.kr; Daehyeon Baek, dhbaek@casys.kaist.ac.kr; Jongse Park, jspark@casys.kaist.ac.kr; Jaehyuk Huh, jhhuh@kaist.ac.kr, KAIST, School of Computing, 291 Daehak-ro, Yuseong-gu, Daejeon, Republic of Korea, 34141.

ACM 1544-3566/2024/3-ART
https://doi.org/10.1145/3653020

and the widening workloads by machine learning have made certain design options that were not commonly used in CPU/GPU computation attractive for custom accelerators.

The first aspect is the matrix-vector multiplication algorithm, which has two possible approaches: dot product and scalar multiplication. Dot product has been the dominant algorithm for software kernels, but accelerators made scalar multiplication a viable option with their customized data flow [1, 5, 18, 26, 42]. The second aspect is the sparse matrix representation, which has two common representative approaches: compressed sparse data format (e.g., CSR and CSC), and bitmap format. In traditional HPC workloads with high sparsity, compressed formats are dominantly used, but sparse ML computation with relatively low sparsity can opt for bitmap formats due to their compact representation capability. With these two design aspects, there are four possible designs.

However, existing SpMV accelerators have exclusively belonged to one of the four design classes. It is yet to be investigated whether a design choice is superior to the others across different workloads and hardware configurations, and why a design choice should be selected for a given workload. Therefore, in addition to optimizing a fixed algorithm and data representation, it is necessary to investigate the possible design options more thoroughly.

With various matrix dimensions and sparsities from HPC workloads and sparse neural network models, this study shows a single fixed design does not always produce the best performance across different architectural configurations, matrices, and input vectors. Dimensions and densities of matrices and vectors affect the efficiency of different sparsity representations, detection mechanisms, and algorithms. Our design space exploration reveals two complementary design options (i.e. compressed sparse format and bitmap format with dot product), one of the two options provides a good performance across a range of input workloads, while neither of them is superior to the other. In addition, a single accelerator needs to support both sparse and dense data effectively, since it is efficient to use the basic dense computation if the input matrix density is relatively high.

To maximize the efficiency of SpMV acceleration under a wide variety of matrix and vector characteristics, this paper proposes a novel triple-mode SpMV accelerator called *Cerberus*, which can change the mode for the best data representation. This paper shows that a single hardware substrate can support dot product-based SpMV computation with two complementary SpMV configurations (i.e. compressed sparse format and bitmap format for sparse matrix representation) and the dense MV computation effectively. Each processing element (PE) contains the indexing mechanism, which supports not only the two computation modes, but also the dense mode. The formats of input matrices are determined based on the characteristics of the input matrix. While the accelerator supports multi-mode operations, to avoid duplicate inputs in different formats, the input matrix is stored in memory using only one format among the three (i.e., compressed sparse, bitmap, and dense).

To support such a multi-mode accelerator, the mode decision process for a given workload is a key technique. This study proposes a selection algorithm, which finds the best mode based on the characteristics of the input matrix without actually running the model on an accelerator. The selection algorithm uses only simple hyperparameters, such as dimension and density, of the input matrix, and suggests which mode is the best.

Instead of multi-mode accelerators, it is possible to use a different accelerator for each workload. Such an approach can be feasible with FPGA accelerators. However, ASIC accelerators with better performance must process a wide variety of workloads with a fixed hardware substrate. Our triple mode support can adaptively handle such various workloads efficiently.

We evaluate the performance with architectural simulation and implemented the hardware in Chisel for behavioral validation and synthesis. We plan to open-source the hardware design after publication. The experimental results from our SpMV accelerator simulation demonstrate that the multi-mode architecture can provide on average 12.1× performance boost over dense-only computation, and 1.5× boost over a fixed best SpMV design across the benchmark workloads and hardware configurations. The prediction model can accurately find the best configuration with 83.3% accuracy, reaching 90% of performance with the oracle model.

Although there have been studies for choosing a good matrix format for GPU SpMV computation [12, 58], to the best of our knowledge, this is the first study to investigate the design space of SpMV accelerators for different

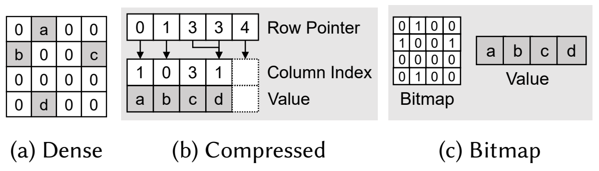


Fig. 1. Data formats for sparse matrix.

algorithms in addition to data representations. Based on the analysis, we propose a performance prediction model and triple-mode accelerator architecture. The main contributions of the paper are as follows:

- This study quantifies the performance implications of algorithm and data representation selection for SpMV accelerators.
- The study finds two complementary design options and proposes Cerberus, a triple-mode accelerator that can process SpMV with the best data representation for a wide range of matrix and vector distributions. In addition, the architecture supports dense mode computation efficiently.
- The study shows that a parameterized model can be used for the best mode selection. The dimensions and densities of matrices with hardware parameters determine the best mode.

## 2 BACKGROUND

### 2.1 Sparse Matrix and Vector Multiplication

A sparse matrix is a matrix that contains a significant number of zero elements, and the sparsity of matrices varies widely depending on applications. The operation $y = Ax$, where $A$ is a sparse matrix and $x$ and $y$ are the dense data and result vectors, is referred to as sparse matrix-vector multiplication (SpMV). SpMV is widely used in high-performance computing and sparse neural networks.

**High-performance computing:** SpMV is a key operation in high-performance computing (HPC) fields such as recommendation systems [41], linear programming [43], graph analytics [7], and big-data analytics [8]. The main bodies of the algorithms use a large number of iterations of matrix-vector multiplication, and vector sparsity can vary during execution [37].

**Sparse neural networks:** Another important SpMV application is the sparse neural network (NN) computation. Researchers use various model compression techniques such as pruning [20], quantization [46], and knowledge distillation [25] to reduce the compute and storage load of neural networks. These techniques optimize the networks by making a subset of model weights to be zeros, resulting in *sparse* neural networks. Sparsity can be further exploited in activations dynamically, but the sparsity of activation is relatively low - making SpMV a major kernel of sparse NN instead of sparse matrix-sparse vector multiplication (SpMSpV). Several recent approaches used MV as the basic building block for NN acceleration with weight matrix and activation vector, both in dense and sparse NN [15, 18].

### 2.2 SpMV Computation

For data format, traditional high-sparsity HPC workloads have mainly used variants of the compressed representation for compression efficiency, and thus there have been many studies to optimize format selection among different compressed formats. However, as machine learning workloads with much lower sparsity are widely used, the bitmap representation has become effective for certain workloads [17]. For algorithms, in CPU and GPU kernels, dot product has been commonly used. However, recent accelerators enabled an efficient custom data flow for scalar multiplication [18]. With the two approaches which have become viable options, it is necessary to investigate their effects on the accelerator design along with the traditional approaches.

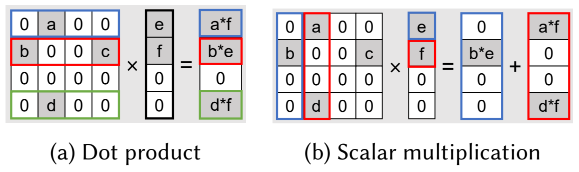


Fig. 2. Algorithms for sparse matrix and vector multiplication.

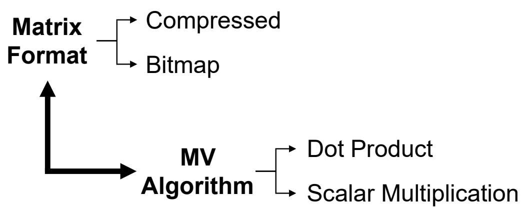


Fig. 3. Two dimensions in the SpMV accelerator design space.

| Name | Matrix Format | MV Algorithm |
|---|---|---|
| EIE [18] | Compressed | Scalar Mult. |
| Tensaurus [60] | Compressed | Dot Product |
| SpaceA [64] | Compressed | Dot Product |
| MASR [17] | Bitmap | Dot Product |
| SIGMA [52] | Bitmap | Dot Product |

Table 1. Design choices of the existing SpMV accelerators.

**Data formats for sparse matrix:** There are several custom data formats to store sparse matrices. Among the variants, the most representative data formats for recent accelerators are the following two: 1) ***Compressed format***, and 2) ***Bitmap format***. Figure 1 illustrates two sparse formats, showing the difference between dense matrix representation (dense format) and sparse formats. The compressed format represents sparse matrices using three 1D arrays: (1) *value* that contains the set of non-zero values in a row or column; (2) *index* that contains the indices of the non-zero values; and (3) *pointer* that constitutes pointers to the initial element of each row or column. In this paper, we cover two most generally-used variants in the compressed format: (1) ***Compressed Sparse Row (CSR)***, and (2) ***Compressed Sparse Column (CSC)*** - a transposed version of CSR. Figure 1b illustrates the CSR format, encoding the position of non-zero elements in a sparse matrix with row pointer and column index array. Note that there exist more variants of compressed format (e.g. coordinate list), and we covered only CSR/CSC since other variants show either very similar functionality with CSR/CSC or are too specified for certain types of data. The bitmap format, on the other hand, represents sparse matrices using (1) *bitmap* where the size is the same as the original matrix and each bit stores the information of whether or not the corresponding data element is zero, and (2) *values* that store the collection of non-zero values.

**Algorithms:** There are two general algorithms to perform SpMV acceleration, as illustrated in Figure 2: 1) ***Dot product***: the values in the result vector are calculated through a set of dot product operations between matrix rows and vector; and 2) ***Scalar multiplication***: the matrix-vector multiplication is split into a series of vector-scalar multiplications. The results of vector-scalar multiplications are a set of vectors, which contain partial outputs that need to be accumulated to produce the final outputs.

| **Matrix Format** | **MV Algorithm** | **Acronym** |
|---|---|---|
| Compressed | Dot Product | $C_mD_a$ |
| | Scalar Mult. | $C_mS_a$ |
| Bitmap | Dot Product | $B_mD_a$ |
| | Scalar Mult. | $B_mS_a$ |

Table 2. Acronyms for SpMV accelerator classes. Subscripts, $m$ and $a$, stand for matrix format and algorithm, respectively.

## 2.3 SpMV Acceleration

Software kernels with CPUs or GPUs mostly assume sparse matrix in compressed format, performing dot product between sparse matrix and dense vector [1, 5, 26, 42, 58]. This is because of the characteristic of traditional HPC workloads: while SpMV with highly-sparse and large matrix has been used widely in HPC. For CPU and GPU execution, dot product generally produces more efficient computation than scalar multiplication. In addition, increasing the variety of sparse workloads expanded the range of density and dimension of a sparse matrix, as well as allowing some vector sparsity in SpMV kernels.

To follow up on such changes in trend, a series of inspiring prior work have proposed customized accelerators for SpMV either with alternative design choices, or variants of conventional CPU/GPU software design choices. In designing the accelerators, the researchers had to navigate the following two dimensions: 1) sparse matrix format and 2) MV algorithm. Figure 3 illustrates the design space and possible design choices. Table 1 lists the five existing accelerators that include SpMV in their targets, placed in the design space.

While these design decisions result in architectural differences and thus significantly affect performance behaviors, there is a lack of study that analyzes the performance implications of these choices. To this end, this paper first analytically and empirically examines various design points in the design space to better understand the implications and trade-offs (Section 3). Through this design space exploration, we observe that there is no single design point that always outperforms others, but rather, the best design choices for SpMV accelerator architecture depend on workload characteristics such as density and weight/vector shapes. Motivated by this insight, we take a hybrid approach and design a multi-mode SpMV accelerator architecture, which prudently selects one of three execution modes that fits best for the given sparse matrix shape and its density level for each SpMV kernel (Section 4).

## 2.4 Problem Scope

Our design space encompasses two sparse matrix formats and two algorithms for SpMV computation as shown in Figure 3. For the input and output vectors, we use the dense format but zeros in the input vector can be dynamically skipped. This study limits its scope to SpMV with dense vectors for HPC workloads or moderately sparse vectors for sparse neural networks, where the dense vector format with dynamic zero skipping is sufficient for efficient computation.

In traditional HPC workloads, SpMV kernels commonly use dense vector formats, while there are a set of applications using very sparse vectors (density < 1%) [23]. Such sparse matrix-sparse vector multiplication (SpMSpV) kernel require an accelerator design specifically optimized for the high sparsity of vectors, which is beyond the scope of this paper. However, emerging sparse NN allows a moderate vector sparsity in their kernel, which shows different characteristics from such conventional SpMSpV: Vectors with moderate sparsity generate dense output vectors when multiplied with sparse matrix, having more similar characteristics to conventional SpMV rather than SpMSpV. This study includes such sparse NN workloads in our target applications.

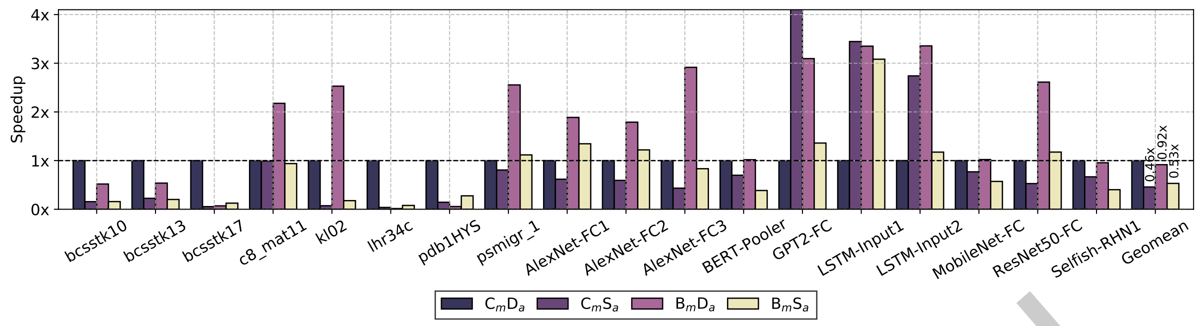


Fig. 4. Speedups for 4 classes, normalized to the best performing one ($C_mD_a$).

## 3 MOTIVATION

### 3.1 Two Aspects of SpMV accelerators

We first architecturally quantify the effect of matrix format and MV algorithm for the SpMV accelerators, navigating the two dimensions illustrated in Figure 3. Table 2 lists the 4 possible designs along with their acronyms: for instance, **C**ompressed **m**atrix - **D**ot product **a**lgorithm is $C_mD_a$**.** In the rest of the paper, each combination is denoted as SpMV accelerator class. Note that when the vector uses sparse format, it uses the same format as the input matrix.

The *sparse matrix format* can be either compressed or bitmap format. One of the major differences between the two formats is the size of metadata (i.e. index information of non-zero elements). While the metadata size of the compressed format grows proportional to the number of non-zero elements, the bitmap format metadata size is fixed by the dimension. For matrices with a high density, the metadata of compressed format is larger than that of bitmap format. Note that we do not consider other specialized sparse formats [6, 36, 59, 60] from our design space because they are optimized for certain types of workloads or architectures.

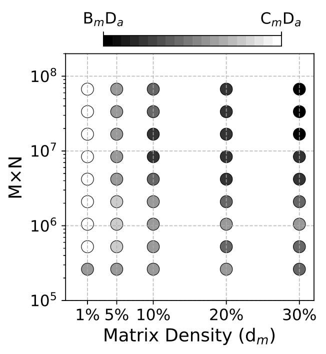
The *algorithm* can be either dot product or scalar multiplication. Dot product-based algorithm repeats dot products between rows of the matrix and the vector to produce a single output element. This requires repetitive reads from the input vector. In addition, unnecessary multiplications should be skipped by the 'index matching' process, which searches the position of non-zero operands to be multiplied. Scalar multiplication, on the other hand, repeats scalar multiplications between non-zero elements of the vector and the associated columns of the matrix to produce partial result vectors. These partial outputs should be merged to produce the final output vector, therefore the output should either be read repetitively or consume extra storage for storing all partial outputs.

### 3.2 Quantifying the Effects of Two Aspects

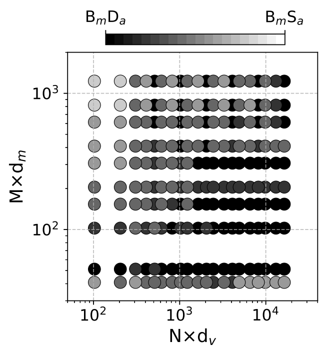
To evaluate all 4 classes, each one is configured to have equal computational resources and on-chip memory. The details of the resource configuration are shown in Table 4 (Section 5.1). The benchmarks include both HPC and ML workloads shown in Section 5.2. In this section, we answer the following two questions with the analysis of 4 configurations:

(1) Which sparse matrix format performs best for SpMV?
(2) Is scalar multiplication or dot product superior to the other? Does the sparse format affect the superiority?

Figure 4 presents the average speedups for the benchmark of Table 5 with 4 classes. The speedups are the normalized performance over the best-performing one ($C_mD_a$). The figure shows that $C_mD_a$ generally outperforms the rest of the classes. However, for a subset of workloads, $B_mD_a$ or $B_mS_a$ significantly outperforms $C_mD_a$.



(a) Matrix format: compressed vs. bitmap ($B_mD_a$ vs. $C_mD_a$).



(b) Algorithm: dot product vs. scalar mult. ($B_mD_a$ vs. $B_mS_a$).

Fig. 5. Preference distribution between two choices of matrix formats (a) and algorithms (b).

**Sparse matrix formats:** The sparse matrix format exhibits trade-offs across different workloads, although the overall best one uses compressed format. Although $C_mD_a$ exhibits the highest overall performance followed by $B_mD_a$ and $B_mS_a$, for a subset of workloads, $B_mD_a$ or $B_mS_a$ significantly outperforms $C_mD_a$. The result shows that a single format does not always provide the highest performance across various workloads. To understand the trade-offs with a broader range of matrix and vector characteristics, we analyze the performance trend of these factors with the microbenchmarks populated using the hyperparameters in Table 6.

Figure 5a shows the correlation between matrix characteristics and sparse matrix formats while fixing the algorithm to the dot-product ($C_mD_a$ vs. $B_mD_a$). The x-axis is a matrix density, and the y-axis is a total number of elements. A darker color indicates a higher performance of $B_mD_a$ over $C_mD_a$, while a lighter color means that $C_mD_a$ has a better performance. The compressed format (i.e. $C_mD_a$) tends to outperform the bitmap format (i.e. $B_mD_a$) in two cases: 1) With a lower matrix density, the size of the compressed format metadata becomes smaller while the bitmap counterpart is constant. This leads to a reduction of memory accesses and an improvement of performance for the compressed format more than the bitmap format. 2) When the total number of elements increases, the required number of leading nonzero detection (LNZD) operations gets larger for the bitmap format. Due to the extra overheads of bitmap format from LNZD operations, the relative performance of the compressed format is improved.

**Algorithms:** For the top-3 classes, two of them use dot-product, while the third one uses scalar multiplication. Figure 5b shows the correlation between the matrix/vector characteristics and algorithms while fixing the matrix format to the bitmap one ($B_mD_a$ vs. $B_mS_a$). The x-axis is the number of non-zero elements in the vector, and the y-axis is the expected number of non-zero elements per the sparse matrix column. A darker color indicates that the performance of $B_mD_a$ is higher than $B_mS_a$, and a lighter color means $B_mS_a$ performs better. The dot product ($B_mD_a$) is better than the scalar multiplication ($B_mS_a$) in more cases. However, $B_mS_a$ performs better than $B_mD_a$ when either the number of non-zero elements per matrix column or the number of non-zero elements in the vector is large. For such cases, repetitive reading of the vector with the dot product incurs higher overheads than the partial sum accesses with the scalar multiplication. We observed the performance trend of microbenchmarks is similar to those of real-world benchmarks, which supports the generality of evaluation with the benchmarks.

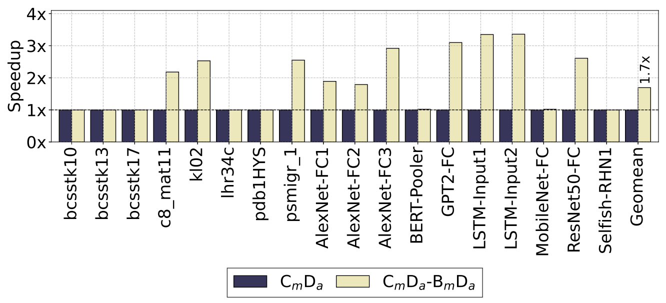


Fig. 6. Performance (speedup) normalized to the single-best one ($C_mD_a$) vs. the per-workload best one between $C_mD_a$ and $B_mD_a$ ($C_mD_a$-$B_mD_a$ Best).

## 3.3 Potential of Multi-mode Acceleration

In the analysis, although the overall best performing one is $C_mD_a$, the next class ($B_mD_a$) outperforms $C_mD_a$ for several workloads. The performance trend of the two modes depends on the input features such as density and dimension of the matrix. Such performance disparity opens a new possibility of multi-mode accelerators which can adapt to input data characteristics. For such multi-mode accelerators, three constraints must be considered. 1) Each supported mode must complement each other across different workloads. 2) Supporting multiple modes should not incur significant extra timing and area overheads. 3) In addition to the mode selection between sparse modes, another important aspect is to support conventional dense matrix-vector multiplication. In sparse matrix-related tasks, not all kernels can benefit from sparsity: for example, a dense layer usually appears in sparse neural network inference tasks. Thus, a single accelerator must be able to process a dense operation too. Note that a full dense matrix-vector multiplication using dot-product is denoted as $D_mD_a$ (Dense matrix - Dot product algorithm) in the rest of the paper.

Considering $C_mD_a$ as the first mode of the multi-mode accelerator, $B_mD_a$ shares the same algorithm and vector format which leads to a similar dataflow. Therefore, we expect that with $B_mD_a$ as a second sparse mode, the hardware complexity for supporting three modes is minimal since the three modes have a similar data flow based on the dot-product algorithm. The difference between the two modes is in the matrix format, which changes zero-skipping only for the matrix, while the vector is fixed to the dense format. Supporting different algorithms incurs a high complexity and requires different logic for zero-skipping depending on compression formats.

Figure 6 presents the potential of selecting between $C_mD_a$ and $B_mD_a$, compared to the best class ($C_mD_a$). The left bar is $C_mD_a$, and the right bar is the better one for each workload between $C_mD_a$ and $B_mD_a$. The result shows that if the better one between $C_mD_a$ and $B_mD_a$ can be used for each workload, it can outperform a single best mode by 69%.

## 4 TRIPLE-MODE ACCELERATION

### 4.1 Overview

As shown in the prior sections, switching among the three modes, $C_mD_a$, $B_mD_a$, and $D_mD_a$, selectively with respect to the workload characteristics would potentially offer higher performance than a fixed best-performing mode. In this section, we propose Cerberus, a triple-mode accelerator architecture design for SpMV with a selection method. The triple-mode accelerator can switch its execution mode between $C_mD_a$ and $B_mD_a$ for sparse computations and also support dense computation. The software selection method determines which mode produces the best performance, purely based on the input characteristics such as matrix dimension and density. Its high accuracy shows that the best mode is highly correlated with the static input characteristics of each SpMV workload.

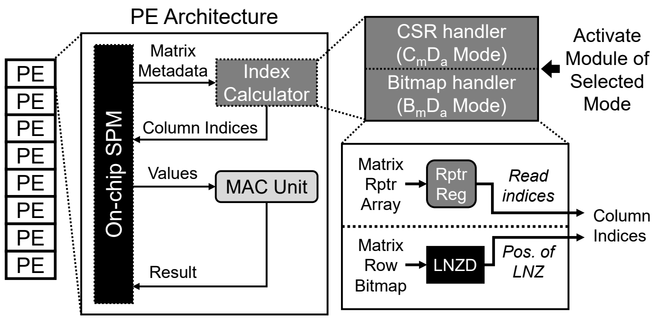


Fig. 7. Overview of Cerberus.

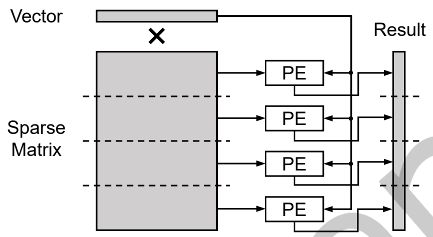


Fig. 8. Data distribution to PE.

Figure 7 illustrates the overall architecture of Cerberus. To maximize the advantages of parallel executions on multiple PEs, we partition the matrix in a row-wise manner to fit in the 1D array of PE, as shown in Figure 8. Each PE is equipped with parallel multipliers and adders. We choose to use 1D PE organization since the 2D alternative requires inter-PE communication due to the row-wise merge operation, which incurs extra performance overheads by extending the critical path without the benefit of data reuse [18]. Unlike dense MMs which can exploit the data reuse feature of a 2D PE array, for SpMV with unstructured and thus irregular sparsity, the 1D organization utilizes PEs more efficiently.

**Mode Selection:** Mode selection is done by the runtime driver software controlling the accelerators. The runtime system uses the dimension and density of the input matrix to determine the mode. The selection result is only dependent on the properties of matrices. For computation with fixed input matrices such as weight matrices for ML inferences, the mode is pre-selected for the matrices, and the matrix is stored in one of the sparse format ($C_mD_a$), bitmap format ($B_mD_a$), or dense format ($D_mD_a$).

For certain applications, an input matrix can be dynamically updated over iterations of MV computation. For such applications, the conversion of the input matrix can occur between compressed and bitmap formats. However, such mode change is very infrequent, since the format selection depends on the density and dimension of the matrix. The dimension does not change for the same matrix, and the density changes only very gradually. In addition, the format conversion requires an update of only the indices or bitmaps without changing the compressed value array. As both formats are row-major, both read and write patterns from the old format to the new one are sequential.

The mode change in hardware causes a negligible overhead because it merely changes the choice of handler module to use. In addition, the mode change occurs only between two different instances of matrix-vector multiplications such as different layers in DNNs. Thus, it does not require any updates on the internal states of PEs. Note that the overhead of mode selection itself is relatively low, as it only requires an offline calculation of polynomial (i.e. performance model of Section 4.3).

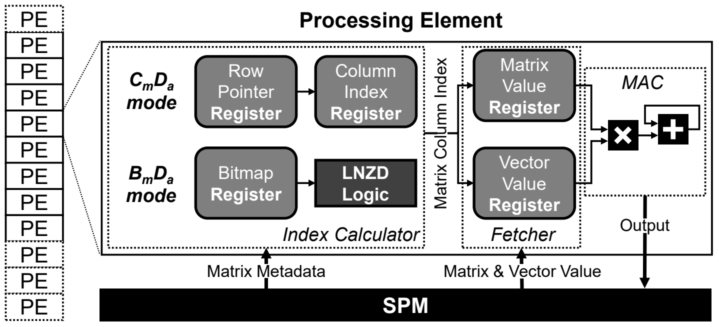


Fig. 9. The internal design of Cerberus PE.

## 4.2 Architecture

Figure 9 shows the detailed architecture of the PE. The intra-PE architecture can be divided into four parts: *on-chip scratchpad memory*, *index calculator*, *value registers*, and *dot product computation unit*.

**On-chip scratchpad memory (SPM):** Each PE contains a small on-chip SPM to store the input and output data. We choose the local SPM instead of the inter-PE shared buffer to minimize the interference between PEs. When the requested data is not in SPM, the PE sends a memory request to the off-chip memory.

**Index Calculator:** Since switching execution mode without significant overheads is the most important requirement for triple-mode architecture, we minimize the change of architecture in mode change by limiting it inside the *index calculator*. The major difference between each mode is a way of calculating the column indices for non-zero values: how to detect the next value to be pushed to the multipliers. As described earlier in Figure 7, the *index calculator* contains two handler modules for $C_mD_a$ and $B_mD_a$ modes. With $C_mD_a$ mode, the *index calculator* first fetches row pointers and detects the position (column index) of non-zero elements contained in the row by reading the column index array sequentially. This requires two 64-bit registers: one for row pointers, and the other for column indices. For $B_mD_a$ mode, the bitmap handler uses a leading nonzero detection logic (LNZD), which sequentially checks the bitmap of a matrix row and returns the position of the next nonzero bit as a column index for the next nonzero value. We use the LNZD design from prior work [34] that enables the LNZD unit to detect nonzero values in a 32-bit window with a delay of 1 cycle.

Also, for efficient nonzero-detection steps, the bitmap is saved in a 512-bit register. For $D_mD_a$ mode, the accelerator can run efficiently by inactivating all components under the *index calculator* and feeding values sequentially. For a mode change, one of the modules is selected as the index calculator and the other module will be deactivated.

**Value Registers (Fetchers):** Using the position information of non-zero values (column indices) produced from the *index calculator*, the matrix and vector values for the corresponding positions are accessed and multiplied for MV operation. For this purpose, two 64-bit registers, matrix value register and vector value register, are used to fetch values from SPM and feed the data to the dot product computation unit.

**Dot Product Computation Unit:** To calculate the output values, the accelerator should execute dot product operations. The *dot product computation unit* is a MAC unit with one multiplier and one adder (accumulator). This single-MAC-per-PE design is sufficient because of two reasons: First, the proposed architecture can exploit parallelism through a PE array. Moreover, as the major performance bottleneck of SpMV is in index calculation, even if there is more than one MAC, the computational unit would not be utilized efficiently.

## 4.3 Software Selector

**Analytical performance model:** Our performance model takes two sets of variables as input, 1) Sparse matrix hyperparameters (matrix dimension and density), and 2) hardware specifications (the number of PEs and SPM size). We use a two-step selection algorithm. The first step decides whether the dense mode ($D_mD_a$) should be considered or not. If the matrix density is higher than the threshold of 87.5% (That is, the threshold of which dense matrix format reaches a footprint smaller than sparse formats), the selector includes the dense mode to the set of possible options, and thus it chooses the best mode out of three modes (Dense, Bitmap, and Compressed; $D_mD_a$, $B_mD_a$, and $C_mD_a$). If the matrix density is lower than 87.5%, $D_mD_a$ becomes very inefficient compared to two other sparse modes, and is excluded from the mode selection, choosing the best one out of only two sparse modes ($B_mD_a$ and $C_mD_a$). We adopted this first step of determining whether $D_mD_a$ should be considered or not, to avoid overly complicating the performance model used in the second step. The performance model estimates the relative delay of each mode, instead of predicting its exact execution cycle.

After the set of possible modes is determined in the first step, the second step evaluates the performance scores of candidates with a performance model. The performance model is essentially a function, which produces a single score calculated by a combination of three estimated metrics: 1) the total time for on-chip memory accesses (*total_onchip_access_time()*), 2) the total time for off-chip memory accesses (*total_offchip_access_time()*), and 3) the total time of non-common additional operations (*total_additional_ops_time()*), as illustrated in Equation 1:

$$\begin{aligned} &\mathit{score = (max(total\_onchip\_access\_time()/P,} \\ &\qquad \mathit{total\_additional\_ops\_time())} \\ &\qquad \mathit{total\_offchip\_access\_time())/E} \end{aligned} \tag{1}$$

| Class | On-chip Access | Additional Operations | Re-Access Window | Off-chip Access |
|---|---|---|---|---|
| $D_mD_a$ | 2×M×N/ value_size/ prefetching_unit + M | M×N - M×N×$d_m$ (Multiplication) | spm_blocksize + 2×N×value_size | {(Re-access window) > spm_capacity} ? {(M×N + N×E + M×value_size)/spm_blocksize} : {(2×M×N×2 + M×value_size)/spm_blocksize} |
| $C_mD_a$ | M + 1 + M×N×$d_m$× (index_size + value_size)/ prefetching_unit + M×N×$d_m$ + M | None | 2×spm_blocksize + N×$d_m$×(index_size + value_size) + value_size×N | {(Re-access window) > spm_capacity} ? {(ptr_size×(M + E) + M×N×$d_m$×(index_size + value_size) + N×E×val_size + M×value_size)/spm_blocksize} : {(ptr_size×(M + E) + M×N×$d_m$×(index_size + value_size) + M×N×value_size)/spm_blocksize} |
| $B_mD_a$ | M×N/(bitmapreg_size×8) + M×N×$d_m$×value_size/ prefetching_unit + M×N×$d_m$ + M | M×N×$d_m$ (LNZD) | spm_blocksize + N/8 + N×$d_m$×value_size + N×value_size | {(Re-access window) > spm_capacity} ? {(M×N/8 + M×N×$d_m$×value_size + N×E×value_size + M×value_size)/spm_blocksize} {(M×N/8 + M×N×$d_m$×value_size + M×N×value_size)/spm_blocksize + M} |

Table 3. Analytical model for the number of three architectural events: 1) on-chip access, 2) off-chip access, and 3) additional integer/bit operations.

$E$ represents the number of PEs, and $P$ is the number of SPM ports per PE. These latencies are calculated by multiplying the estimated amount of each architectural event and the statically collected cycle count number to perform the event on the hardware. The details for calculating the estimated amount (number) of each architectural event is available in Table 3.

As the zero-skipping overhead for vectors is ignorable for SpMV, the vector density information is unnecessary - which makes the offline mode selection feasible. Once the model calculates the score using the algorithmic and

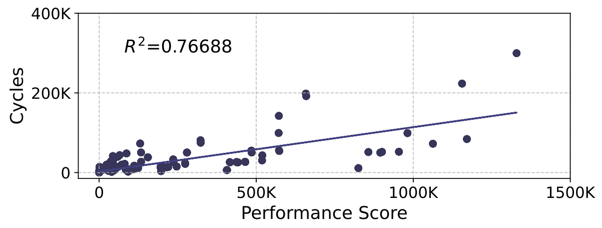


Fig. 10. Correlation between the model-estimated score and the real performance in cycle counts.

hardware-based parameters, the model can determine which mode would perform better for the given workload. For $C_mD_a$ mode, *total_additional_ops_time()* is always 0, while $B_mD_a$ (and $D_mD_a$) gets non-zero value. $B_mD_a$ mode requires leading non-zero detection (LNZD) for matrix bitmaps and the overhead is equal to the number of matrix non-zero values for $B_mD_a$ mode.

**Effectiveness of the performance model:** We evaluate the reliability of the performance model with the microbenchmarks of Table 6. Figure 10 reports the correlation between the calculated score by the performance model and the performance in cycle counts collected through simulation experiments. For the entirety of 600 microbenchmarks, we notice a clear trend line, which corroborates the effectiveness of the model. The $R^2$ coefficient score is 0.77 which is also consistent with the above observation. Although the model-estimated score and the performance of Figure 10 is not perfectly linear, the performance model is still fair enough as the performance model is only required for decision by simply comparing the performance score of each mode (i.e. selecting the smallest value among two or three candidates). The accuracy of the model-based software selector is 79.8%, and the misprediction occurs when the performance gap between the two modes are low: thus, with the model-based software selector, the multi-mode accelerator reaches 92.2% of the oracle performance.

## 5 EVALUATION

### 5.1 Methodology

**Simulation infrastructure:** We use an in-house cycle-accurate simulator, which thoroughly models the intra-accelerator operations and interactions of hardware components. We use this simulator to accurately estimate the execution runtime of operations. Table 4 specifies the simulation parameters, which are largely based on prior work [29, 56]. We use SRAM for on-chip scratchpad memories and HBM2 for off-chip memory. The number of multipliers/adders per PE and the number of read/write ports of SPM per PE are equivalent, for efficient parallel processing. We use CACTI to calculate the optimal number of ports, which does not cause the access latency increment. We use the same set of benchmarks used in Section 3.

**Baselines:** The baseline architecture is a dense MV accelerator ($D_mD_a$) using the same configuration as the sparse counterpart. We also compare our architecture with NVIDIA RTX3090 GPU using cuBLAS [48] and cuSPARSE library [47] for dense/sparse matrix-vector multiplication, respectively. For a fair comparison with GPU, we exclude data transfer time between the host and the device memory. Note that we select GPU routines that perform single-precision floating point operation since it was the lowest precision available for cuBLAS and cuSPARSE. While there exists several prior work accelerating SpMV, it is yet infeasible to compare their performance with our proposed architecture directly due to the following reasons: First, most prior work did not release their simulator in public, therefore only limited comparison with a subset of our evaluation benchmark set was feasible. While one of the prior work, SIGMA [52] open the simulator [45] in public, since the simulator only supports the matrix density range between 1% and 100%, which is still a subset of our evaluation set. Because of this, we only contain the comparison with SIGMA in the latter subsection (i.e. Section 5.6 considering the design scope of SIGMA.

| Configurations | Parameters |
|---|---|
| **Processing Elements** | |
| Array Configuration | $256 \times 1$ |
| Bitmap Register Capacity | 64B |
| Integer Multiplication Latency | 1 cycle |
| Integer Addition Latency | 1 cycle |
| LNZD Latency | 1 cycle |
| Compute Precision | INT16 |
| **On-Chip SPM** | |
| # of Ports | 4 |
| Capacity | 16KB/PE |
| Access Latency | 1 cycle |
| **Off-Chip Memory** | |
| Bandwidth | 600GB/s |
| Access Latency | 100 cycles |
| Chip Frequency | 1GHz |

Table 4. Simulator configurations.

## 5.2 Benchmarks

**Real-world benchmarks:** For the design space exploration, we use 18 real-world SpMV workloads collected from various domains. Table 5 shows the 18 SpMV workloads collected from a wide range of HPC applications and sparse neural networks. We collect HPC workloads from SuiteSparse [9] by selecting sparse matrices used in different domains as prior work [64] did. The HPC benchmark names are the name of each matrix specified in SuiteSparse. On the other hand, for sparse neural network kernels, we either perform the model compression using existing pruning tools [32, 68] over the pre-trained dense models or take the pre-pruned sparse NNs from prior work [19, 38]. The NN-based benchmarks are represented using the base neural network name attached with the used layer name at the end. The columns, $M$, $N$, $d_m$, and $d_v$, represent the number of rows, the number of columns, the density of the matrix, and the density of vectors. As explained in Section 2.2, some sparse NN workloads contain vector sparsity for fair evaluation.
**Microbenchmarks:** While we use real-world workloads as the primary evaluation means, their limited variety prevents us from thoroughly examining the entire design space and analyzing the performance characteristics. To remedy this limitation, we synthetically create a set of microbenchmarks sweeping through a wide range of density levels and tensor shapes. The ranges of these hyperparameters are listed in Table 6 and values are derived from the observed hyperparameters from Table 5. As the hyperparameters, $M$, $N$, $d_m$, and $d_v$, have 4, 6, 5, and 5 variations, the total number of microbenchmarks is 600 (=$4 \times 6 \times 5 \times 5$).

## 5.3 RTL Implementation

To verify the design, we model the proposed architecture in RTL. We implement on-chip components in Chisel [57] to generate Verilog implementation and perform behavioral simulation as well as post-synthesis timing simulation of Xilinx Vivado v2021.2 [2]. Our component-level RTL implementation includes on-chip components described 4.2 and the controller logic. In addition to component-level implementation, we also implement the full on-chip execution flow for each mode (i.e. $C_mD_a$, $B_mD_a$, $D_mD_a$) in Chisel, verify the functionality with Vivado behavioral simulation, and perform synthesis. Table 7 summarizes the required number of cells, I/O ports, and nets for RTL

| Name | Problem Domain | M | N | $d_m$ | $d_v$ |
|---|---|---|---|---|---|
| bcsstk10 | Structural Prob. [13] | 1,086 | 1,086 | 0.02 | 1.0 |
| bcsstk13 | Computational Fluid Dynamics [13] | 2,003 | 2,003 | 0.02 | 1.0 |
| bcsstk17 | Structural Prob. [13] | 10,974 | 10,974 | 0.004 | 1.0 |
| c8_mat11 | Combinatorial Prob. [14] | 4,562 | 5,761 | 0.09 | 1.0 |
| kl02 | Linear Programming Prob. [43] | 71 | 36,699 | 0.08 | 1.0 |
| lhr34c | Chemical Process Simulation [43] | 35,152 | 35,152 | $6\times10^{-4}$ | 1.0 |
| pdb1HYS | Weighted Unidirected Graph [62] | 36,417 | 36,417 | 0.002 | 1.0 |
| psmigr_1 | Economic Prob. [13] | 3,140 | 3,140 | 0.06 | 1.0 |
| **Name** | **NN Model** | **M** | **N** | $\mathbf{d}_m$ | $\mathbf{d}_v$ |
| AlexNet-FC1 | AlexNet [33] | 4,096 | 9,216 | 0.09 | 0.30 |
| AlexNet-FC2 | | 4,096 | 4,096 | 0.09 | 0.38 |
| AlexNet-FC3 | | 1,000 | 4,096 | 0.25 | 0.46 |
| BERT-Pooler | BERT-Small [11] | 768 | 768 | 0.20 | 1.0 |
| GPT2-FC | GPT2-Small [53] | 50,257 | 768 | 0.13 | 1.0 |
| LSTM-Input1 | LSTM [68] | 6,000 | 1,500 | 0.3 | 0.3 |
| LSTM-Input2 | | 6,000 | 1,500 | 0.3 | 1.0 |
| MobileNet-FC | MobileNet [28] | 1,000 | 1,280 | 0.28 | 0.66 |
| ResNet-FC | ResNet50 [21] | 1,000 | 2,048 | 0.31 | 0.83 |
| Selfish-RHN1 | Selfish RNN [38] | 1,660 | 830 | 0.15 | 1.0 |

Table 5. Real-world SpMV benchmarks.

| Hyperparameters | Values |
|---|---|
| M | 512, 1,024, 2,048, 4,096 |
| N | 512, 1,024, 2,048, 4,096, 8,192, 16,384 |
| $d_m$ | 0.01, 0.05, 0.1, 0.2, 0.3 |
| $d_v$ | 0.2, 0.4, 0.6, 0.8, 1.0 |

Table 6. Hyperparameters for microbenchmarks.

implementation of a single PE for each design choice. $C_mD_a$ and $B_mD_a$ require more resources than $D_mD_a$ for handling metadata, and $B_mD_a$ require additional cells and nets for LNZD logic.

## 5.4 SpMV Kernel Speedup

**Real-world benchmark evaluation:** Figure 11 reports the speedup of Cerberus compared to the dense MV accelerator baseline and three alternative choices: (1) GPU running dense MV routine (`cublasSgemv()`) of cuBLAS library, (2) GPU running SpMV routine (`cusparseSpMV()`) of cuSPARSE library, and (3) $C_mD_a$ (i.e.

| Arch | Cells | I/O Ports | Nets |
|---|---|---|---|
| $D_mD_a$ | 114 | 68 | 1804 |
| $C_mD_a$ | 261 | 100 | 3367 |
| $B_mD_a$ | 316 | 100 | 3189 |

Table 7. RTL Design Information.

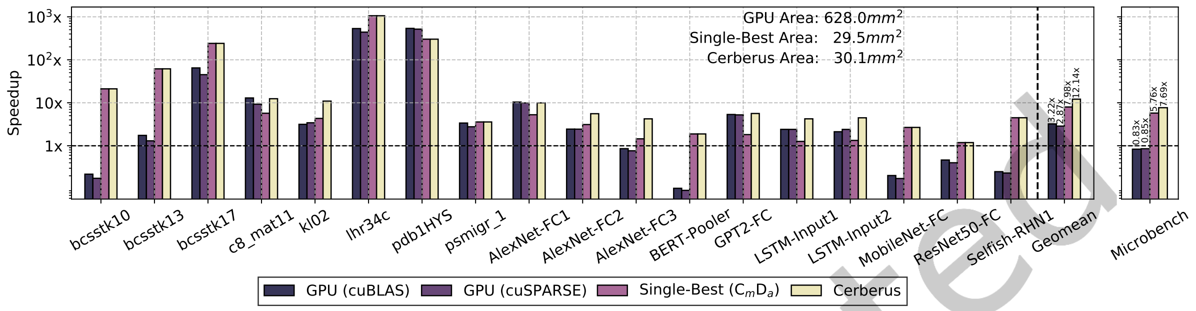


Fig. 11. Speedup of the Cerberus normalized to the dense MV accelerator, in log-scale.

single-best), the best-performing fixed-mode SpMV accelerator architecture. The results show that Cerberus achieves 12.1× speedup on average, while cuBLAS, cuSPARSE, and single-best only obtain 3.2×, 2.9× and 8.0× of average speedup, respectively. In other words, the triple-mode acceleration of Cerberus unlocks 4.2× and 1.5× "additional" speedup on top of what the two alternative accelerators offer. In comparison with the multi-mode accelerator with an oracle selector (i.e. software selector with 100% accuracy), Cerberus shows 90% performance of the oracle multi-mode accelerator. This is incurred by the misprediction of 4 cases out of 18 benchmarks, of which are the cases where performance gap between two sparse modes is low. This performance boost of Cerberus can be translated into the throughput of 134.8GOPS on average. GPU still shows relatively low performance in the view of throughput: cuBLAS reaches 35.8GOPS, and cuSPARSE only reports 31.9GOPS on average. Note that in a few cases, GPU outperforms Cerberus (e.g., pdb1HYS). However, such performance gain is originated from the considerably larger area overhead of GPUs (i.e., GPU (628$mm^2$) vs. Cerberus (30$mm^2$)) in addition to the higher clock GPU clock frequency (i.e. GPU (1395MHz) vs Cerberus (1000MHz)). Therefore, in the perspective of cost-effectiveness and energy efficiency, Cerberus significantly outperforms GPUs, achieving better performance with 1/20 of the GPU area.

**Larger SpMV workloads:** In addition to the SpMV workloads of Table 5, we evaluate Cerberus with larger real-world SpMV workloads. We collect four large sparse matrices from SuiteSparse [9]: `cant`, `consph`, `mac_econ_fwd500`, and `shipsec1` of which dimension is in range of 62,451 to 206,500 and density is in range of 0.003% to 0.05%. All of these additional benchmarks prefer $C_mD_a$, showing several orders of magnitudes of speedup above Dense MV baseline, and 100% accuracy of software selector. This is because of the characteristics of the sparse matrix: a large sparse matrix with low density prefers $C_mD_a$, following the trend of Figure 11.

**Microbenchmark evaluation:** To evaluate the effectiveness of our triple-mode acceleration with Cerberus, we also use 600 microbenchmarks of Table 6, the results of which are shown at the rightmost corner of Figure 11. From the results, we observe a similar trend to the results of the real-world benchmarks, showing that Cerberus offers 9.0× (=7.69/0.85) and 1.3× (=7.69/5.76) additional speedup compared to GPU and $C_mD_a$, respectively.

To better visualize the benefits of Cerberus, we present the speedup results of all the evaluated microbenchmarks as a cumulative distribution function (CDF), as delineated in Figure 12. The relative speedup of $10^0$ (i.e., 1×)

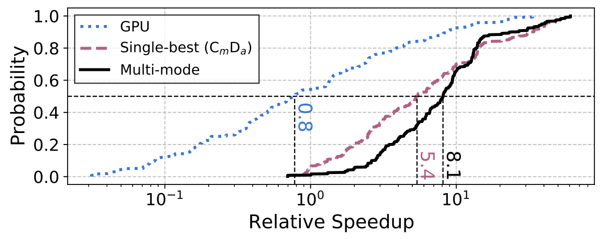


Fig. 12. CDF of speedup with GPU, single-best, and Cerberus for microbenchmarks, normalized to the dense accelerator.

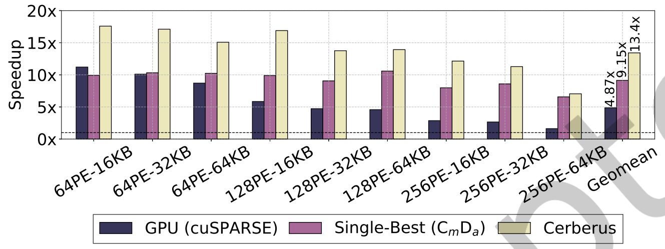


Fig. 13. Speedup of Cerberus with different configurations compared to the dense accelerator baseline.

means that the corresponding accelerator offers the same performance as the dense one. Note that the black line (i.e. Cerberus) is located mostly on the right side of the $10^0$ point. On the other hand, the blue line (i.e., GPU) reaches the $10^0$ point at roughly the probability of 0.5, which means that roughly 50% of microbenchmarks experience lower performance than the dense baseline. Also, while the red dotted line (i.e. Single-best) is also located mostly on the right side of the $10^0$ point, the more steep slope of the black line shows that Cerberus performs better than $C_mD_a$ in most cases. These results, along with the average results reported in Figure 11, suggest that the proposed triple-mode accelerator not only offers higher performance on average but also covers a wider range of SpMV operations than the other alternatives.

## 5.5 Sensitivity to Hardware Configurations

The proposed mode selector is designed in such a way that captures various hardware resource constraints of the triple-mode accelerator and determines the execution mode. To demonstrate the effectiveness of the mode selector, we perform a sensitivity study that examines the performance implications as the hardware constraints are altered. For the experiment, we modify the number of PEs from 256 to 128 and 64, and the size of per-PE SPM from 64KB to 16KB and 32KB.

Figure 13 shows the speedup of the proposed architecture as we alter the hardware configurations. Note that the baseline is the dense accelerator using the given hardware resource for each configuration. The results are geometrically averaged across the 18 benchmarks. We observe that for all hardware configurations, Cerberus outperforms single-best, which is translated to on average 13.4× higher speedup. However, depending on the configurations, the benefits of triple-mode support vary significantly. Cerberus outperforms the single best mode ($C_mD_a$) significantly when the SPM size is small. When the on-chip memory size increases from 16KB to 64KB, the gap between the single-best and Cerberus decreases. The trend shows that Cerberus can provide high performance without requiring a large on-chip memory which requires a high static power consumption.

| Model | Layers | Avg. Weight Density | Avg. Act. Density |
|---|---|---|---|
| AlexNet [33] | 5 Conv, 3 FC | 0.11 | 0.59 |
| LSTM [68] | 2 LSTM | 0.3 | 0.82 |
| MobileNet [28] | 14 Conv, 1 FC | 0.48 | 0.60 |
| Selfish RNN [38] | 10 RHN, 1 FC | 0.47 | 0.99 |
| SqueezeNet [30] | 2 Conv, 8 Fire block | 0.33 | 0.78 |

Table 8. End-to-end sparse neural network models. RHN is a variant of RNN layer and a fire block is a CNN layer block that constitutes 3 convolutions.

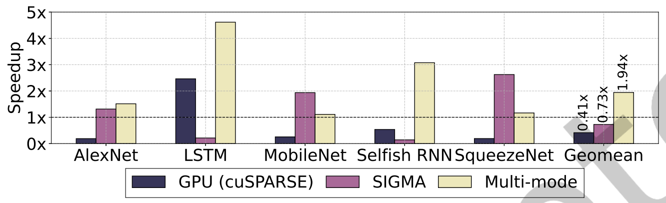


Fig. 14. End-to-end speedup results over dense MV accelerator for five different sparse neural network models.

## 5.6 Case Study: Sparse Neural Network

**Methodology:** As discussed in Section 2, the proposed architecture can handle not only separated SpMV workloads, but also complex sparse matrix-related benchmarks such as full neural networks. To demonstrate the capability and its effectiveness, we use a set of sparse neural network models, as listed in Table 8. The third and fourth columns represent the average density of all weight parameters and input activations. Note that we map convolution layers to sparse matrix multiplication using the im2col operation and further break them down into a set of SpMV operations to map them on the SpMV accelerators, while evaluating GPU with `cusparseSpGEMM_compute` for fair comparisons. We also employ STONNE [45] simulator using the equivalent set of configurations in SIGMA [52], for comparison with prior work.

**End-to-end speedup results:** Figure 14 reports the speedup comparison results of GPU, SIGMA, and Cerberus, with the dense accelerator as the baseline. On average, our Cerberus accelerator offers 1.9× speedup, while GPU and SIGMA experience 59% and 27% of slowdown. While we observe the average speedup gains, we notice that different neural networks see significantly different performance behaviors. When the neural network has low weight density (i.e. AlexNet and LSTM), Cerberus provides substantial performance gain since it can efficiently exploit the weight sparsity. When the SpMV is a major kernel of the neural network (i.e. LSTM and Selfish RNN), Cerberus significantly outperforms SIGMA by handling SpMV more efficiently. On the other hand, for the neural network that largely consists of convolution layers with irregular SpGEMM/SpMM (i.e. MobileNet and SqueezeNet), SIGMA performs better than ours. This is due to the approach of handling SpGEMM kernel: While Cerberus requires the decomposition of SpGEMM kernel into multiple SpMV kernels, SIGMA can compute SpGEMM with exploiting the locality of both operands (i.e. input and filter matrices).

While the performance gains fluctuate as the layer types and hyperparameters vary, Cerberus promises superior performance on average, which demonstrates the fact that the proposed accelerator solution is not just optimized for single SpMV, but also can be applied for complex applications such as end-to-end sparse neural network inferences.

**Layer-wise breakdown:** Figure 15 shows the speedup of Cerberus in comparison with GPU, $C_mD_a$, and SIGMA, running sparse AlexNet [19]. The baseline is the dense accelerator, and Cerberus provides 1.5× speedup. In other

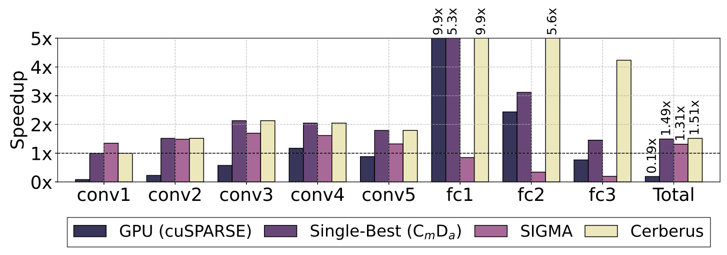


Fig. 15. Layer-wise speedups over the dense MV accelerator for each of the AlexNet's layers. 'Total' represents the end-to-end speedup result.

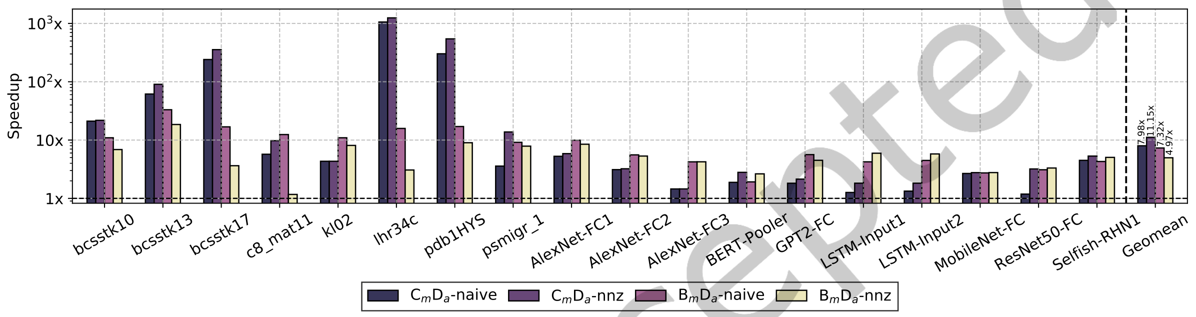


Fig. 16. Dimension-aware partitioning vs. nnz-aware partitioning for sparse modes of Cerberus, normalized to the dense MV accelerator.

words, Cerberus offers 7.9× speedup over GPU, and is 15% faster than SIGMA. Among 5 SpGEMM layers (i.e. conv1-conv5) of AlexNet, Cerberus outperforms SIGMA in 4 layers by selecting the best SpMV accelerator mode for each kernel. Likewise, Cerberus shows notable performance improvements over SIGMA in 3 SpMV layers (i.e. fc1-fc3) through the multi-mode architecture, resulting in 15% speedup over SIGMA in total. In comparison with $C_mD_a$, however, our accelerator results in only 2%p performance gain in total (i.e. end-to-end execution). This is because of the characteristic of AlexNet: the software selector of Cerberus selects $C_mD_a$ mode for convolution layers and $B_mD_a$ for fully-connected layers, and the effect of three fully-connected layers is almost ignored in end-to-end execution time due to their relatively fast execution time. Note that the choice of software selector is nearly optimal, and Cerberus reported 97.4% performance with respect to the triple-mode accelerator with oracle selector. We expect that for larger sparse neural networks with more variety in layer dimension and density, Cerberus will report even better end-to-end performance.

## 5.7 Data Partitioning and Load Balancing

**Matrix partitioning:** Basically, Cerberus splits sparse matrices to distribute an equivalent number of rows in each PE. However, such dimension-aware partitioning scheme may result in an imbalance in the amount of data in each PE, when the distribution of non-zero values is far from a uniform distribution. To resolve the potential threat of load imbalance between PE in dimension-aware partitioning, we apply an alternative partitioning scheme: nnz-aware (non-zero aware) partitioning for sparse modes in Cerberus. The nnz-aware partitioning scheme splits the matrix to distribute an equivalent number of non-zero elements in each PE, within a granularity of rows.

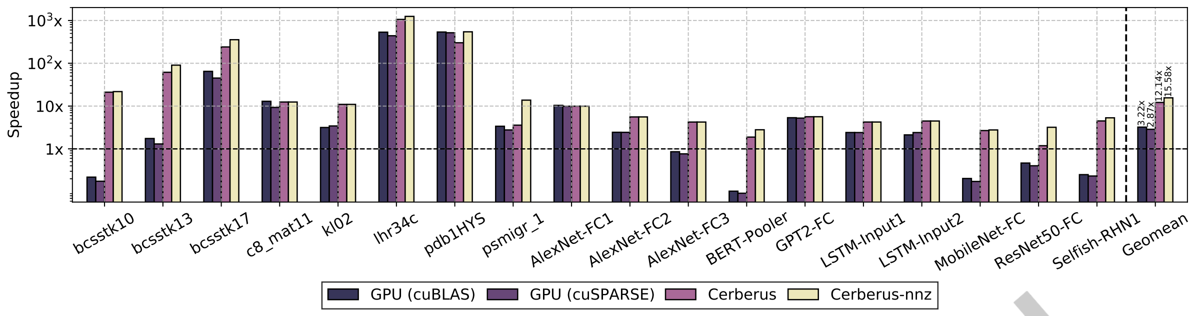


Fig. 17. Speedup of Cerberus with best-partitioning scheme, normalized to the dense MV accelerator, in log-scale.

| **Components** | **Area** ($mm^2$) | | **Power (mW)** | |
|---|---|---|---|---|
| | **Single-Best** | **Cerberus** | **Single-Best** | **Cerberus** |
| SPM | 29.2 | 29.2 | 13978.6 | 13978.6 |
| LNZD Logic | - | 0.055 | - | 0.61 |
| Registers | 0.12 | 0.73 | 505.8 | 3034.6 |
| Adders | 0.052 | 0.052 | 2.13 | 2.13 |
| Multipliers | 0.15 | 0.15 | 16.70 | 16.70 |
| Total | 29.5 | 30.1 (+2.2%) | 14503.2 | 17032.7 (+17.4%) |

Table 9. Area and power results of the single-best SpMV accelerator design ($C_mD_a$) and Cerberus.

**The effect of data partitioning on the performance:** Figure 16 compares the performance of the dimension-aware partitioning and nnz-aware partitioning for each sparse mode in Cerberus, normalized to the dense MV accelerator. Basically, SpMV workloads with non-zero distribution far from uniform distribution show a larger gap between dimension-aware partitioning and nnz-aware partitioning (e.g. `bcsstk17`, `BERT-Pooler`). However, the preference for partitioning scheme differs between two sparse modes: $C_mD_a$ always prefer (i.e. always shows better performance) nnz-aware partitioning. This is because the execution time of $C_mD_a$ is determined by the memory access for non-zero values and metadata. $B_mD_a$, however, prefers nnz-aware partitioning only when the matrix density is relatively high. Since the time consumed for 32-bit window LNZD dominates the execution time of $B_mD_a$, when the matrix density is lower than 1/32 (i.e. the expected number of non-zero values within 32-bit window gets lower than 1), the performance of $B_mD_a$ relies on matrix size (i.e. the total number of elements) rather than the number of non-zeros. This results in the relatively low performance of nnz-aware distribution in $B_mD_a$, making $B_mD_a$ prefer dimension-aware partitioning in general.
**Cerberus with best-partitioning:** Based on the analysis for different trend of data partitioning between two sparse modes, we apply best-partitioning scheme for each sparse mode of Cerberus in Figure 17 (i.e. Cerberus-nnz; nnz-aware partitioning for $C_mD_a$, dimension-aware partitioning for rest of modes). By selecting the best partitioning scheme for each mode to decrease loss from load imbalance, Cerberus-nnz achieves 28% performance improvement over naïve Cerberus with dimension-aware partitioning-only. This performance improvement relies on two advantages of nnz-aware partitioning: Performance boost of $C_mD_a$ and correction on misprediction of the performance model.

### 5.8 Area and Power Overheads

Table 9 shows the breakdown of the area overhead and power consumption of the single-best and Cerberus when the 256 PEs and 16KB SPM configurations are used. For this experiment, we use 28nm technology and 1GHz clock frequency. We measure the area and power results for on-chip scratchpad memories (SPMs) using CACTI, while for other hardware components, we use the reported numbers from several prior work [4, 34, 35, 55]. The results show that most of the on-chip resources are consumed by SPMs, which are in charge of the majority of operations in SpMV, and the rest impose small area/power overheads. Compared to the single-best accelerator, Cerberus only imposes 2.2% and 17.4% additional area and power overheads, respectively, which are marginal considering the significant performance benefits achievable by triple-mode support.

### 5.9 Discussion

**Extension to other sparse kernels:** Cerberus is an accelerator designed for SpMV operation. However, the performance model-based multi-mode design with multiple data formats and a single algorithm can be applied generally to sparse-dense tensor multiplication operation, by either decomposing the tensor operation into SpMV or extending the Cerberus architecture and performance model to higher dimensions. Note that the sparse-sparse tensor multiplication of which both tensors have high sparsity (e.g. SpGEMM for graphs and scientific problems) would require a revised performance model and architecture that considers both tensors' sparsity.

## 6 RELATED WORK

**SpMV accelerators:** Many accelerator architectures have been proposed for SpMV. EIE [18] is one of the initial studies in accelerating SpMV in neural networks. Tensaurus [60] regards SpMV as a subset of a larger problem domain; general sparse tensor - dense tensor multiplication. Two-Step [54] concentrates on utilizing memory bandwidth with a divide-and-conquer strategy for SpMV. SpaceA [64] overcomes the limitation of memory bandwidth with processing-in-memory architecture for SpMV. MASR [17] focuses on acceleration of specific RNN inference using a bitmap to encode inputs of SpMV. SIGMA [52] accelerates nonsquare SpGEMM for sparse neural network training, treating SpMV as a subset of their problem domain.
**Software frameworks for SpMV acceleration:** For CPU or GPU computing, compressed (CSR) matrix-dot product algorithm is a common combination for SpMV. Intel MKL [31] is a software library for CPUs, while cuSPARSE [47] and bhSPARSE [39] work as a software framework for GPUs. These libraries support various operations with sparse matrices including SpMV. Furthermore, Sedaghati et al. [58] and Dinkelbach et al. [12] introduce the automated sparse matrix representation selector for SpMV with ML-based decision model which is applicable for GPU SW.
**Other sparse accelerators:** Researchers have been focusing on hardware acceleration for tensor algebra with sparse matrix. OuterSPACE [49], Sparse-TPU [22], SpArch [67], MatRaptor [59], and InnerSP [3] focus on the acceleration of SpGEMM for HPC applications. ExTensor [23] is a general sparse tensor algebra accelerator, and Qin et al. [51] accelerates sparse tensor - dense tensor multiplication with support for multiple sparse formats. Flexagon [44] proposes a flexible SpGEMM accelerator that supports multiple dataflow for sparse neural networks. SpAtten [61] and Sanger [40] handle sparse attention layers and transformers based on the sparse matrix multiplication acceleration. SparTen [16], SCNN [50], UCNN [24], GoSPA [10], and Cambricon-X [66] all accelerate sparse CNN inference. Procrustes [65] deals with the training of sparse CNN. Sparseloop [63] provides a framework for selecting the best hardware architecture for sparse CNN inference.

## 7 CONCLUSION

This study investigates the impact of sparse matrix and vector multiplication algorithms and sparse matrix data formats for the accelerator design. The investigation shows that no single algorithm and data format can produce optimal performance with various workloads and hardware resources. This paper finds the two complementary modes for sparse matrix-vector multiplication, which provides good performance for a range of workloads. With the prediction model for the best mode selection, this paper proposes Cerberus, a triple-mode accelerator design that can reconfigure itself for two different sparse modes and the base dense operation mode.

## REFERENCES

[1] Mohammad Almasri and Walid Abu-Sufah. 2020. CCF: An efficient SpMV storage format for AVX512 platforms. *Parallel Comput.* 100 (2020), 102–710.

[2] AMD. 2021. Xilinx Vivado 2021.2. https://www.xilinx.com/products/design-tools/vivado.html.

[3] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. InnerSP: A Memory Efficient Sparse Matrix Multiplication Accelerator with Locality-aware Inner Product Processing. In *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 116–128.

[4] P. Balasubramanian, D. L. Maskell, and N. E. Mastorakis. 2020. Speed, energy, and area optimized erly output quasi-delay-insensitive array multipliers. *PLoS ONE* 15, 2 (2020), 1–20.

[5] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *SC '09: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[6] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11.

[7] Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. 2013. Scalable Matrix Computations on Large Scale-Free Graphs Using 2D Graph Partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.

[8] Aric Coady. 2004. Process and system for sparse vector and matrix representation of document indexing and retrieval. US Patent 6,751,628.

[9] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[10] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. 2021. GoSPA: An Energy-efficient High-performance Golbally Optimized SParse Convolutional Neural Network Accelerator. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*. 1110–1123.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[12] Helge Ülo Dinkelbach, Badr-Eddine Bouhlal, Julien Vitay, and Fred H. Hamker. 2022. Auto-Selection of an Optimal Sparse Matrix Format in the Neuro-Simulator ANNarchy. *Frontiers in Neuroinformatics* 16 (2022).

[13] Iain Duff, Roger Grimes, and John Lewis. 1989. The original Harwell-Boeing collection. *ACM Trans. Math. Software* 14, 1 (1989), 1–14.

[14] Jean-Guillaume Dumas. 2017. SIMC: Sparse Integer Matrix Collection. (2017). https://doi.org/10.18709/PERSCIDO.2017.11.DS185

[15] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*. 1–14.

[16] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolution Neural Networks. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO)*. 151–165.

[17] Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe, Alexander M Rush, Gu-Yeon Wei, and David Brooks. 2019. MASR: A Modular Accelerator for Sparse RNNs. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 1–14.

[18] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. 243–254.

[19] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *Proceedings of the International Conference on Learning Representations (ICLR)* (2016), 1–14.

[20] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]

[22] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices. In *The Proceedings of the International Conference on Supercomputing (ICS)*. 1–12.

[23] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO)*. 319–333.

[24] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. 2018. UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*. 674–687.

[25] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Destilling the Knowledge in a Neural Network. arXiv:1503.02531 [stat.ML]

[26] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Cataly urek, Srinivasan Parthasaranthy, and P. Sadayappan. 2018. Efficient Sparse-Matrix Multi-Vector Product on GPUs. In *HPDC '18: The 27th International Symopsium on High-Performance Parallel and Distributed Computing*. 66–79.

[27] Seongmin Hong, Seungjae Moon, Junsoo Kim, Sungjae Lee, Minsub Kim, Dongsoo Lee, and Joo-Young Kim. 2022. DFX: A Low-latency Multi-FPGA Appliance for Accelerating Transformer-based Text Generation. In *Proceedings of the 55th International Symposium on Microarchitecture (MICRO)*. 616–630.

[28] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolution Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV]

[29] Bongjoon Hyun, Youngeun Kwon, Yujeong Choi, John Kim, and Minsoo Rhu. 2020. NeuMMU: Architectural Support for Efficient Address Translations in Neural Processing Units. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1109–1124.

[30] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. arXiv:1602.07360 [cs.CV]

[31] Intel. 2019. Intel Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html.

[32] Alexander Kozlov, Ivan Lazarevich, Vasily Shamporov, Nikolay Lyalyushkin, and Yury Gorbachev. 2020. Neural Network Compression Framework for fast model inference. arXiv:2002.08679 [cs.CV]

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of Neural Information Processing Systems (NeurIPS)*. 1–9.

[34] Kumarasamy Kunaraj and R. Seshasayanan. 2013. Leading one detectors and leading one position detectors - an evolutionary design methodology. *Canadian Journal of Electrical and Computer Engineering* 36, 3 (2013), 103–110.

[35] Po-Yu Kuo, Ming-Hwa Sheu, Chang-Ming Tsai, Ming-Yan Tsai, and Jin-Fa Lin. 2021. A Novel Cross-Latch Shift Register Scheme for Low Power Applications. *Applied Sciences* 11, 1 (2021), 1–11.

[36] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage, and Analysis*. 1–15.

[37] Min Li, Yulong Ao, and Chao Yang. 2020. Adaptive SpMV/SpMSpV on GPUs for Input Vectors of Varied Sparsity. (2020). arXiv:2006.16767 [cs.DC]

[38] Shiwei Liu, Decebal Constantin Mocanu, Yulong Pei, and Mykola Pechenizkiy. 2021. Selfish Sparse RNN Training. In *Proceedings of the 38th International Conference of Machine Learning*. 1–12.

[39] Weifeng Liu and Brian Vinter. 2015. Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors. *Parallel Comput.* 49, C (2015), 179–193.

[40] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A Co-Design Framework for Enabling Sparse Attention using Reconfigurable Architecture. In *Proceedings of the 54th International Symposium on Microarchitecture (MICRO)*. 977–991.

[41] Xin Luo, MengChu Zhou, Shuai Li, Zhuhong You, Yunni Xia, and Qingsheng Zhu. 2015. A Nonnegative Latent Factor Model for Large-Scale Sparse Matrices in Recommender Systems via Alternating Direction Method. *IEEE Transactions on Neural Networks and Learning Systems* 27, 3 (2015), 579–592.

[42] Duane Merrill and Michael Garland. 2016. Merge-Based Parallel Sparse Matrix-Vector Multiplication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[43] Csaba Mészáros. 2004. Linear programming problems from C. Mészáros. http://www.sztaki.hu/~meszaros/public_ftp/lptestset.

[44] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. 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 (ASPLOS 2023)*. 252–265.

[45] Fransico Muñoz-Matrínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2021. STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 122–125.

[46] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. 2021. A White Paper on Neural Network Quantization. arXiv:2106.08295 [cs.LG]

[47] M. Naumov, L. S. Chien, P. Vandermersch, and U. Kapasi. 2022. CUSPARSE Library. https://developer.nvidia.com/cusparse.

[48] NVIDIA. 2023. cuBLAS. https://developer.nvidia.com/cublas.

[49] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 724–736.

[50] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Kecler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. 27–40.

[51] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, Sudarshan Srinivasan, Dipankar Das, Gordon E. Moon, Sivasankaran Rajamanickam, and Tushar Krishna. 2021. Extending Sparse Tensor Accelerators to Support Multiple Compression Formats. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. 1014–1024.

[52] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vinnet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *Proceedings of the 26th International Symposium on High Performance Computer Architecture (HPCA)*. 58–70.

[53] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://github.com/openai/gpt-2

[54] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO)*. 347–358.

[55] Vikas K. Saini, Shamin Akhter, and Tanuj Chauhan. 2016. Implementation, Test Pattern Generation, and Comparative Analysis of Different Adder Circuits. *VLSI Design* 2016 (2016), 1.

[56] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN Accelerator Simulator. arXiv:1811.02883 [cs.DC]

[57] Martin Schoeberl. 2019. *Digital Design with Chisel*. Kindle Direct Publishing.

[58] Naser Sedaghati, Te Mu, Louis-Noël Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM on International Confernece on Supercomputing (ICS '15)*. 99–108.

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

[60] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed sparse-Dense Tensor Computations. In *Proceedings of the 26th International Symposium on High Performance Computer Architecture (HPCA)*. 689–702.

[61] Hanrui Wang, Zhekai Zhang, and Song Han. 2020. SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 97–110.

[62] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 1–12.

[63] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2022. Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling. In *Proceedings of the 55th International Symposium on Microarchitecture (MICRO)*. 1377–1395.

[64] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *Proceedings of the 27th International Symposium on High Performance Computer Architecture (HPCA)*. 570–583.

[65] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. 2020. Procrustes: a Dataflow and Accelerator for Sparse Deep Neural Network Training. In *Proceedings of the 53rd International Symposium on Microarchitecture (MICRO)*. 711–724.

[66] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An Accelerator for Sparse Neural Networks. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*. 1–12.

[67] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 261–274.

[68] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. 2019. Neural Network Distiller: A Python Package For DNN Compression Research. arXiv:1910.12232 [cs.LG]