Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing

Francisco Muñoz-Martínez
Universidad de Murcia (Spain)
francisco.munoz2@um.es

Raveesh Garg
Georgia Tech (USA)
raveesh.g@gatech.edu

Michael Pellauer
NVIDIA (USA)
mpellauer@nvidia.com

José L. Abellán
Universidad de Murcia (Spain)
jlabellan@um.es

Manuel E. Acacio
Universidad de Murcia (Spain)
meacacio@um.es

Tushar Krishna
Georgia Tech (USA)
tushar@ece.gatech.edu

ABSTRACT

Sparsity is a growing trend in modern DNN models.

Existing Sparse-Sparse Matrix Multiplication (SpMSpM) accelerators are tailored to a particular SpMSpM dataflow (i.e., Inner Product, Outer Product or Gustavson's), which determines their overall efficiency. We demonstrate that this static decision inherently results in a suboptimal dynamic solution. This is because different SpMSpM kernels show varying features (i.e., dimensions, sparsity pattern, sparsity degree), which makes each dataflow better suited to different data sets.

In this work we present Flexagon, the first SpMSpM reconfigurable accelerator that is capable of performing SpMSpM computation by using the particular dataflow that best matches each case. Flexagon accelerator is based on a novel Merger-Reduction Network (MRN) that unifies the concept of reducing and merging in the same substrate, increasing efficiency. Additionally, Flexagon also includes a new L1 on-chip memory organization, specifically tailored to the different access characteristics of the input and output compressed matrices. Using detailed cycle-level simulation of contemporary DNN models from a variety of application domains, we show that Flexagon achieves average performance benefits of 4.59×, 1.71×, and 1.35× with respect to the state-of-the-art SIGMA-like, SpArch-like and GAMMA-like accelerators (265%, 67%, and 18%, respectively, in terms of average performance/area efficiency).

CCS CONCEPTS

• **Hardware** → **Hardware accelerators**; **Emerging architectures**.

KEYWORDS

Deep Neural Network Accelerators, Sparse-Sparse Matrix Multiplication, Dataflow, Merger-Reduction Network, Memory Hierarchy

ACM Reference Format:
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, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3582016.3582069

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023
ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582069

Accelerator	Architectural Features	IP	OP	Gust
TPU [12]	Dense Systolic Array	N/A	N/A	N/A
SIGMA [28]	Configurable Reduce Tree	✓	✗	✗
ExTensor [10]	Intersection Unit	✓	✗	✗
MatRaptor [30]	Merger	✗	✗	✓
Gamma [34]	Fiber Cache, Merger	✗	✗	✓
Outerspace [23]	Merger	✗	✓	✗
SpArch [35]	Matrix condenser, merger	✗	✓	✗
Flexagon (This Work)	Flexible Merge/Reduce tree and memory controller	✓	✓	✓

Table 1: Comparison of Flexagon with prior Sparse DNN accelerators in terms of supported dataflows. IP=Inner Product, OP=Outer Product, Gust=Gustavson's (Row-wise Product).

1 INTRODUCTION

Sparsity in tensors is an emerging trend in modern DNN workloads [19, 21, 29]. These workloads have diverse sparsity ratios, ranging from 0.04% to 90%, and are used in various applications, ranging from personalized recommendations [21] to Natural Language Processing [3]. Sparsity in weights stems from pruning [9] and sparsity inside activations stems from nonlinear functions such as ReLU. As a result, exploiting the benefits of sparsity by directly implementing sparse matrix-matrix multiplication (SpMSpM) has become an important target for customized DNN accelerators [10, 23, 25, 28, 30, 34, 35].

The most common way for these accelerators to exploit sparsity is by using compressed formats like Bitmap, CSR, and CSC to store and operate (multiply and accumulate) only the non-zero values. This allows for a significant reduction in both the memory footprint and the number of operations, which in turn translates into significant energy savings. However, these accelerators vary widely in their hardware implementation and in the exploited dataflow. The dataflows used by these accelerators in terms of the loop order of computation have been broadly classified into `Inner Product` (`IP`), `Outer Product` (`OP`) and Row-wise-Product, often called `Gustavson's` (`Gust`) [7].

Table 1 shows prior sparse accelerators and the dataflows they support. While state-of-the-art sparse accelerators such as SIGMA [28], SpArch [35] and GAMMA [34] have been optimized for a fixed dataflow (`IP`, `OP` and `Gust`, respectively), in this paper, we make the important observation that *the optimal dataflow changes from*

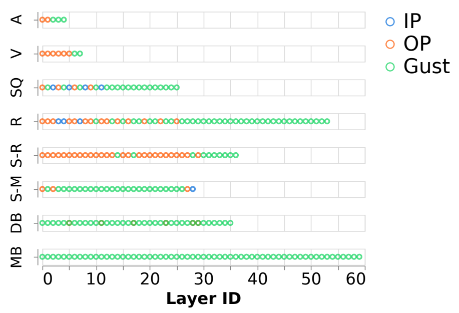


Figure 1: Dataflow that obtains the best performance per layer across some DNN models (see Table 2 that includes their sparsity ratios). IP=*Inner Product*, OP=*Outer Product* and Gust=*Gustavson's*.

a DNN model to another, and even within a DNN model, from one layer to another, so that contemporary fixed-dataflow accelerators cannot adapt well to maximize DNN application performance.

To back up our observation, Fig. 1 shows the dataflow that obtains the best performance per layer given the execution of 8 entire DNN models obtained from MLPerf benchmark suite [29] as well as some extra models (details in Table 2). Observe that we consider heterogeneous models from different domains, sizes, and sparsity ratios. For *MB*, we only show the first 60 layers, which represent 20% out of the total number of layers. To model the three dataflows, the executions have been performed on a 64-Multiplier SIGMA-like, SpArch-like and GAMMA-like architectures (further details in Section 4). The NLP models *DB* and *MB* present a clear trend towards `Gust`. On the other hand, extremely sparse models, such as *S-R* and *V*, benefit from `OP` in 73% and 75% of the layers, respectively. The rest of the DNN models present a high variability across layers, and the most efficient dataflow changes given the different features of each layer. This highlights that one dataflow does not fit all, and so there is an opportunity to increase efficiency via dynamic adaptation of the architectural components to the most suitable dataflow.

The value of supporting flexible dataflows has been explored extensively for dense DNNs [2, 15, 16, 24]. However, support for flexible dataflow acceleration for sparse workloads is much more challenging because of the different ways in which these accelerators handle sparsity. For example, the `IP` dataflow implemented in SIGMA [28] implements a reduction network called FAN to *reduce* the generated partial sums at once, as well as the capacity to perform *intersections* to execute a sparse dot product. On the contrary, the `OP` and `Gust` dataflows implemented in accelerators like SpArch [35] and GAMMA [34] produce partial sums instead of complete sums, and hence, require *merging* the non-zero partial sums and use merger trees for this purpose. A naive implementation using separate hardware widgets for reductions and merges would lead to significant area overhead (see Section 5.3).

To efficiently support different SpMSpM workloads to run modern sparse DNNs, we present *Flexagon*, the first (to our knowledge) reconfigurable sparse and homogeneous DNN accelerator that can be dynamically adapted to execute the most suited SpMSpM dataflow on a per DNN layer basis. Flexagon features a novel unified *Merger-Reduction Network (MRN)* that supports both reductions of dot products and the merging of partial sums. We propose a tree-based topology where the nodes are configured to act either as accumulators or comparators, as explained in Section 3. Flexagon also features a new L1 on-chip memory organization composed of three customized memory structures that are able to capture the memory access pattern of each dataflow. The first memory structure is a simple read-only FIFO, which is designed for the sequential accesses that occur during some stages in the three dataflows. The second one is a low-power cache used to back up the random accesses caused mainly by the `Gust` dataflow. Finally, a customized memory structure called PSRAM is specifically designed to store and read psums, which is essential for both `OP` and `Gust` dataflows. These memory structures allow us to support all the three dataflows with minimal area and power overheads. Further, our accelerator also prevents the hardware from requiring explicit expensive conversions of compression formats (i.e., from CSR to CSC or vice-versa) [26] between layers as it is possible to easily switch among the most convenient dataflow given a particular compression format (details discussed in Section 3).

We summarize our key contributions:

(1) We demonstrate that each SpMSpM operation in modern sparse DNN layers presents different memory access patterns according to matrix dimensions and sparsity patterns. As a consequence, the dataflow that maximizes the performance of a particular SpMSpM operation not only can change between DNN models, but also from layer to layer within a particular DNN model.

(2) We present a new inter-layer dataflow mechanism that enables compression format conversions without explicit hardware modules.

(3) We design Flexagon, which hinges on a novel network topology (called MRN) that allows, for the first time, support for the three dataflows, and a new L1 on-chip memory organization to effectively capture the memory access patterns that each dataflow exhibits for input, output, and partial sums.

(4) We extensively evaluate Flexagon using cycle-level simulations of several contemporary DNN models from different application domains, and RTL implementation of its principal elements. Our results demonstrate that Flexagon achieves average performance benefits of 4.59× (ranges between 2.09× and 7.41×), 1.71× (ranges between 1.04× and 4.87×), and 1.35× (ranges between 1× and 2.13×) with respect to the state-of-the-art SIGMA-like, SpArch-like and GAMMA-like accelerators (265%, 67%, and 18%, respectively, in terms of average performance/area efficiency).

2 BACKGROUND

2.1 Compression formats

Following the same taxonomy used in ExTensor [10], the SpMSpM operation computes the operation $C_{M,N} = A_{M,K} \times B_{K,N}$, where the three matrices are 2-dimensional tensors. Since these matrices are typically sparse (see Table 2), they are compressed to encode the non-zero values while preserving the computation intact (lossless compression) [14]. In our work, we focus on the widely used unstructured compression formats CSR and CSC. A matrix encoded in CSR format employs three 1-dimensional tensors to store the non-zero values in a row-major data layout: a data vector to represent

DNN	Appl	nl	AvSpA	AvSpB	AvCsA	AvCsB	MinCsA	MinCsB	MaxCsA	{MaxCsB	Cycles(10^6) CPU
Alexnet (A)	CV	7	70	48	0.56	13.6	0.02	0.18	1.01	63.41	3804
Squeezenet (S)	CV	26	70	31	0.05	1.54	0.001	0.02	0.58	26.6	2751
VGG-16 (V)	CV	8	90	80	0.55	2.90	0.02	0.15	10.42	0.90	6012
Resnets-50 (R)	CV	54	89	52	0.19	1.30	0.001	0.007	1.0	26.64	4185
SSD-Resnets (S-R)	OR	37	89	49	0.12	3.60	0.003	0.003	10.1	0.50	6429
SSD-Mobilenets (S-M)	OR	29	74	35	0.16	0.31	0.002	0.0004	1.0	1.65	5379
DistilBERT (DB)	NLP	36	50	0.04	2.25	0.35	1.12	0.23	4.5	0.94	5748
MobileBERT (MB)	NLP	316	50	11	0.10	0.07	0.03	0.003	0.125	0.01	4893

Table 2: DNN models used in this work. Appl=Application domain (CV=Computer Vision, OR=Object Recognition, NLP=Natural Language Processing), nl=Number of layers, AvSp{A,B}=Average sparsity of the matrices A and B (in %), AvCs{A,B}=Average compressed matrix size for the matrices A and B (in MiB), MinCs{A,B}=Minimum compressed matrix size for the matrices A and B (in MiB), MaxCs{A,B}=Maximum compressed matrix size for the matrices A and B (in MiB), Cycles(10^6) CPU = Number of cycles obtained after running the benchmarks using MKL in a CPU system.

the non-zero values, a row pointer vector to store the index position where each row begins within the data vector, and a column index vector to store the column of each non-zero value. Similarly, the CSC uses a column-major data layout: a data vector, a column pointer vector to store the index position of the start of a column, and a row index vector to store the row index of each non-zero data value. Observe that both CSR and CSC employ the same compression method, and thus, can be seen as a single compression format. This is important as an accelerator would use the same control logic needed to handle both of them. This facilitates the implementation of the control logic (further details in Section 3.5) in our accelerator.

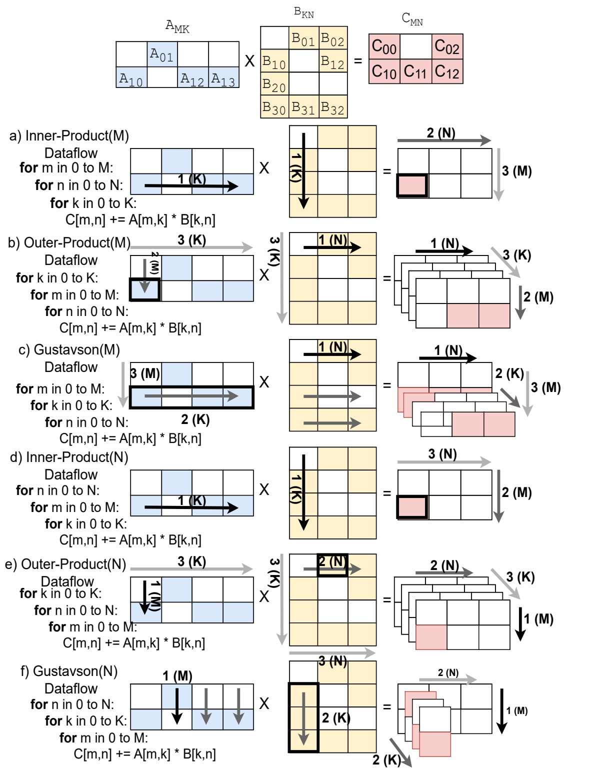
As in previous works (e.g. [34]), we will use the term *fiber* to denote each compressed row or column. Each fiber contains a list of duples (coordinate, value), sorted by coordinate. We use the term *element* to refer to one duple in the fiber.

2.2 SpMSpM dataflows

SpMSpM operation is based on a triple-nested for-loop that iterates over A's and B's independent dimensions *M* and *N*, and co-iterates over their shared dimension *K*. Depending upon the level of the co-iteration in the loop nesting, *three different dataflows* have been identified for SpMSpM computation: IP (co-iteration at the innermost loop), OP (co-iteration at the outermost loop) and Gust (co-iteration at the middle loop). Additionally, these dataflows result in *six possible variants* according to how the independent dimensions (*M* and *N*) are ordered for each of them (two variants per dataflow). Notice that each variant favors the stationarity of one of the dimensions (the outermost one) over the other. This way, we distinguish each variant by (M) if the computation is *M*-stationary or (N) if it is *N*-stationary. Fig. 2 shows the resulting six dataflow variants. Each dataflow defines how the elements flow during execution, and thus, the opportunities for data reuse. Table 3 gives a detailed taxonomy of each approach, which we summarize as follows:

Inner Product (IP): A single full sum at a time is generated, with no merging of partial sums. This requires a hardware intersection unit to align effectual inputs.

Outer Product (OP): A single input scalar at a time is consumed, generating many partial sums (matrices). This requires intersection between the stationary fiber and every element of the fiber of the other operand. Also, it requires merging hardware of the output partial matrices, possibly leading to extra memory traffic.



```
a) Inner-Product(M) Dataflow
for m in 0 to M:
  for n in 0 to N:
    for k in 0 to K:
      C[m,n] += A[m,k] * B[k,n]
b) Outer-Product(M) Dataflow
for k in 0 to K:
  for m in 0 to M:
    for n in 0 to N:
      C[m,n] += A[m,k] * B[k,n]
c) Gustavson(M) Dataflow
for m in 0 to M:
  for k in 0 to K:
    for n in 0 to N:
      C[m,n] += A[m,k] * B[k,n]
d) Inner-Product(N) Dataflow
for n in 0 to N:
  for m in 0 to M:
    for k in 0 to K:
      C[m,n] += A[m,k] * B[k,n]
e) Outer-Product(N) Dataflow
for k in 0 to K:
  for n in 0 to N:
    for m in 0 to M:
      C[m,n] += A[m,k] * B[k,n]
f) Gustavson(N) Dataflow
for n in 0 to N:
  for k in 0 to K:
    for m in 0 to M:
      C[m,n] += A[m,k] * B[k,n]
```

Figure 2: Dataflow combinations for matrix multiplication. For simplicity, non-compressed (dense) matrices are shown. We use the term X(D) in the matrices to express loop ordering (X) and traversed dimension (D).

Gustavson's (Gust): A single input at a time is consumed, but only to generate partial sums into the current fiber. This allows the intersection to be done in *leader-follower* style, where the effectual coordinates of the stationary tensor retrieve entire fibers of the other operand. This requires merging hardware, but only into the current fiber rather than entire matrices.

For the rest of the paper, we will pedagogically use *M*-stationary dataflows during the explanations, although everything would apply for the *N*-stationary dataflows as well.

Dataflow	Informal Name	Stationary Tensor	Stationary Fiber	Streaming Tensor	A format	B format	C format	Intersection	Merging
MNK	Inner Product(M)	C	A	B	CSR	CSC	CSR	Scalar A vs Scalar B	N/A
KMN	Outer Product(M)	A	B	C	CSC	CSR	CSR	Fiber A vs Fiber B	Tensor C
MKN	Gustavson's(M)	A	C	B	CSR	CSR	CSR	Scalar A vs Fiber B	Fiber(M)
NMK	Inner Product(N)	C	B	A	CSR	CSC	CSC	Scalar B vs Scalar A	N/A
KNM	Outer Product(N)	B	A	C	CSC	CSR	CSC	Fiber B vs Fiber A	Tensor C
NKM	Gustavson's(N)	B	C	A	CSC	CSC	CSC	Scalar B vs Fiber A	Fiber(N)

Table 3: Taxonomy of dataflow properties. Traversal order is given outermost-to-innermost in loop order.

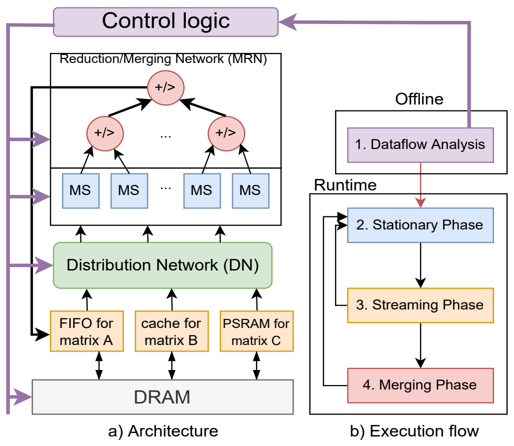


Figure 3: Flexagon high-level overview.

3 FLEXAGON DESIGN

Fig. 3a shows a high-level overview of the architecture of the Flexagon accelerator. As observed, Flexagon consists of a set of multipliers, adders and comparators, as well as three on-chip SRAM modules specifically tailored to the storage needs of matrices A, B and C for the three SpMSpM dataflows. In addition, in order to allow for the highest flexibility, all the on-chip components are interconnected by using a general three-tier reconfigurable network-on-chip (NoC) composed of a Distribution Network (DN), a Multiplier Network (MN), and a Merger-Reduction Network (MRN), inspired by the taxonomy of on-chip communication flows within AI accelerators [16]. These components are controlled by the control unit which is configured by the mapper/compiler before the execution.

Flexagon's execution phases are shown in Fig. 3b. The process begins with a dataflow analysis (phase 1), which is carried out offline. Here, a mapper/compiler examines the features of the SpMSpM operation to be executed (i.e., matrix dimensions and sparsity patterns) and decides the dataflow (between the six available described in Section 2) that best matches the operation, generating the tiling scheme and the particular values for the signals that configure the operation of the accelerator for the rest of the phases.

The next three phases are performed during runtime according to these generated signals and are repeated several times according to the number of execution tiles. The first runtime phase is called **stationary phase** (phase 2), which delivers data that will be kept stationary in the multipliers to reduce the number of costly memory accesses. According to the dataflows description presented in Section 2 for *M*-stationary dataflows, this stationary data belongs to matrix A, while matrix B is streamed during the **streaming phase** (phase 3). For *N*-stationary dataflows this happens in the reverse order. These two phases generalize for the three dataflows. The **merging phase** (phase 4) is only necessary for both OP and Gust dataflows and is the one in charge of merging the fibers of partial sums that have been previously generated during the streaming phase. This phase is skipped in the IP dataflow as no merging is required.

In this work, we focus our attention on the accelerator design as well as on the way the three phases operate in order to give support to the six possible dataflows (three SpMSpM dataflows, two variants, M or N-stationary, each) over the same hardware substrate. We leave the study of the tool required for dataflow analysis, tiling selection and generation of the configuration file for the accelerator (phase 1 in the Offline part in Fig. 3b) for future work.

3.1 On-chip Networks

One of the main novelties of Flexagon is its ability (through proper configuration) to support the six dataflows described in Section 2 using the same hardware substrate. To do so, the accelerator is equipped with a three-tier configurable NoC able to adapt to the communication features of each dataflow. Next, we describe each subnetwork in detail:

Distribution network (DN): This module is used to deliver data from the SRAM structures to the multipliers. In order to enable the high flexibility that the three SpMSpM dataflows require, the DN needs to support unicast, multicast and broadcast data delivery. To achieve this, and at the same time ensure high energy efficiency, we utilize a Benes network similar to previous designs like SIGMA [28]. This network is an N-input, N-output non-blocking topology with $2 \times log(N) + 1$ levels, each with N tiny 2×2 switches.

Merger-Reduction network (MRN): Previous designs like MAERI [16] or SIGMA [28] have used specialized tree-based reduction networks (RNs) such as ART or FAN to enable non-blocking reduction of multiple clusters of psums. These RNs provide high flexibility for the IP dataflow as its purpose is to reduce a cluster of psums. In the case of OP and Gust dataflows, other works such as [34, 35] employ a tree-based topology to perform the merge operation of the psums once they are generated. In our design, we have, for the first time, unified this concept, and have designed a merger-reduction network able to both reduce and merge psums. Figure 4a shows the microarchitecture overview of a 16-wide MRN. As it may be observed, similar to previous designs, we also employ an augmented tree-based topology because this is the fastest way to perform both the reduction and merging operations. Different from previous designs, the MRN topology augments the nodes with comparators and switching logic able to exchange the mode of operation (see Figure 4b). This allows us to perform both operations while keeping low area and power overheads (details in Section 5.3) and, as we describe later, enables direct support for the three SpMSpM dataflows. Furthermore, we employ a connection with two links between the nodes, allowing the MRN to traverse not only data values but also the coordinates needed in both OP and Gust dataflows. The selection of the configuration is done by the mapper/compiler, which generates the control signals that feed the configuration logic module (Control Logic in Fig. 3a) of the accelerator, which in turn routes the appropriate signals to the nodes, configuring its operation modes according to the dataflow and layer dimensions.

Multiplier network (MN): Similar to other designs such as MAERI, this network is composed of independent multipliers that can operate in two different modes: i) *Multiplier mode*: the unit performs a multiplication and sends the result to the MRN. This mode is used during the entire execution when the IP dataflow is configured, and during the streaming phase when either the OP or Gust dataflows are configured; ii) *Forwarder mode*: the multiplier forwards directly the input, which is typically a psum, to the MRN. As we will clarify in the examples presented next, this mode is essentially configured during the merging phase in both the OP and Gust dataflows. The microarchitecture of the multipliers is depicted in Figure 4c.

3.2 Walk-through Examples

Next, we illustrate how Flexagon works when running the three dataflows for the multiplication of matrices A and B from Fig. 2, considering the runtime phases explained earlier. We pedagogically assume the IP(M), OP(M) and Gust(M) dataflows. Note that, the IP(N), OP(N) and Gust(N) dataflows could be executed in the same manner by exchanging matrices A and B. To ease the explanation, we assume a simple 4-multiplier accelerator, and we walk through the activity of the three sub-networks. In the explanation, we mention the on-chip SRAM modules needed for storing matrices A, B, C and psums (see the yellow boxes in Fig. 3b). Section 3.4 provides an in-depth description of these memory structures.

3.2.1 *Example of Inner-Product dataflow.*

Fig. 5 shows the IP(M) dataflow. In the figure, we represent with "*" the psums that need to be reduced by the adders in the tree to produce the final values for matrix C.

Stationary phase: First, during the stationary phase, the controller maps as many fibers of matrix A (i.e., rows of A) as possible to the multipliers, reading all the elements sequentially from the dedicated SRAM structure called FIFO for matrix A. Each cluster of multipliers will perform the dot product operation.

Streaming phase: After filling the multipliers with the fibers of A, the controller multicasts each fiber of matrix B (i.e., each column) to the configured clusters in the MN. To do so, the controller uses the row coordinate of each element in the fiber of B to detect whether it intersects with the column coordinate in the fiber of A. If this happens, the value is sent out to the corresponding multiplier.

3.2.2 *Example of Outer-Product dataflow.*

Fig. 6 shows the same example as before but now assuming the OP(M) dataflow. We also show the customized SRAM structure for C called PSRAM that is utilized for storing the psums for matrix C. As we will explain in Section 3.4, this structure stores blocks of elements (coordinate, value).

Stationary phase: During the stationary phase, the fibers of matrix A (i.e., columns of A) are delivered to the multipliers sequentially following the CSC compression format. In our particular case, the four multipliers store the elements $A_{1,0}$, $A_{0,1}$, $A_{1,2}$, and $A_{1,3}$.

Streaming phase: During the streaming phase, each multiplier keeps stationary an element $A_{m,k}$, given m in the range $[0,M)$ and k in the range $[0,K)$, in order to linearly combine the non-zero elements $B_{k,0:N-1}$, generating a psum fiber where all the elements share the row (m) and a particular k iteration (i.e., the partial matrix where these elements belong to). Consecutive multipliers generating psums for different rows for the same k iteration, do not need the psum to be merged together. Thus, the generated psums must be sent out to the SRAM structure, in order to be merged in a third phase. Also, since multiple rows can run in parallel, the PSRAM 's set is indexed by rows. Furthermore, since the number of non-zeros in matrix A is not known a priori, it might happen that multiple fibers from matrix A may fit in a single iteration, causing that multiple partial outputs for the same row, but for different k iterations, may run in parallel. Since the number of psums for a particular row and for a particular k iteration is not known at runtime, we must assign static space in the PSRAM to store the psums from different k iterations that may be running in parallel and being kept in the same row (i.e., set). To do so, we divide each set in the PSRAM in blocks (i.e., lines), and each line contains a valid bit to indicate the validity of the data, a k value, indicating the k iteration that belongs to that group of partial sums and the block of data. By doing this, each line can hold, at a particular time, psums for different k iterations for a particular row. This way, if the number of psums for a particular iteration exceeds the line size, it may use any free line in the corresponding set of the PSRAM (not necessarily a consecutive one). The details about the organization and operation of the PSRAM are given in Section 3.4.

In the example of Fig. 6, we see three steps regarding the streaming phase. In the first step, the controller sends the first element of the four fibers (across the K-dimension) to its corresponding multiplier. For example, the first multiplier which keeps stationary the element $A_{1,0}$ receives the first element of the fiber for the row (i.e., iteration k) 0. In step 2, each multiplier generates a psum (indicated by the symbol *), which is the first element for the 4 fibers generated across the K-dimension. These psums are then stored in the PSRAM . The first psum $^*C_{1,1}$ is allocated in set 1, as it is indexed by its row coordinate. Use of sets allows us to execute multiple rows in parallel. Then, since the first line is free, the psum is stored there, enabling the valid bit and indicating that the element belongs to *K0*. Dividing rows into blocks allows holding psums corresponding to different K for a particular row. The second psum, $^*C_{0,0}$ is allocated to the set 0 (its row coordinate) and since the first line is here, the cache enables the valid bit and tags the line with *K1*. The last two elements share coordinates (i.e., s $^*C_{1,0}$), but belong to a different partial matrix (*K2* and *K3*). These two elements go to the same set in the PSRAM but to different lines, each tagged with its iteration k (i.e., *K2* and *K3*). This allows locating the psum fibers in the correct order during the merging phase.

In step 3, the second set of elements for the four fibers is produced, following the same execution scheme. For the sake of brevity, we do not show how the last element from the longest psum fiber (i.e., fiber K3) is produced, and directly show the contents of the PSRAM

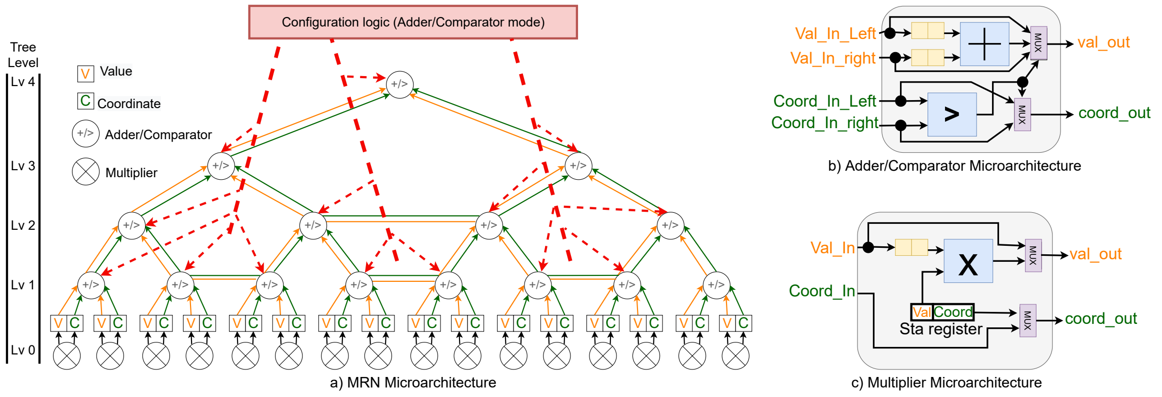


Figure 4: a) MRN topology. b) architecture of the MRN's nodes (Adder/Comparator nodes). c) architecture of Multipliers.

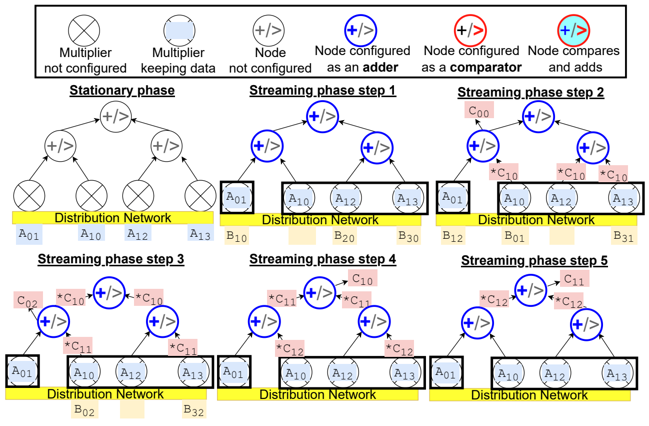


Figure 5: Example of Flexagon running SpMSpM using an `Inner-Product(M)` dataflow. "*" indicates psums.

just before starting the merging phase (merging phase step 1). We can see in the PSRAM figure from the merging phase step 1 that the element has been stored in the last line within the first set, as the third line is already full.

Merging phase: The merging phase proceeds row by row. For each row, the controller fetches the elements for the different k-iteration fibers from the PSRAM . These elements are stored in different lines and can be identified by their tags, consuming the elements and sending them to the MRN in order to be merged. Each unit in the MRN compares the column coordinate (i.e., the N-dimension). If the coordinates match, then the values of the elements are accumulated. Otherwise, the node sends up the tree the element with the lowest coordinate. The last two rows in Fig. 6 show 8 merging steps. The 4 first steps (Merging phase step 1 to step 4) merge the first row. In the second row, there are 3 psum fibers ready to be merged. In step 5, the first elements for the three fibers ($K0$, $K2$ and $K3$) are sent to the MRN. In step 6, the psums $^*C_{1,1}$ and $^*C_{1,0}$ compare their column coordinate. Since they do not match, and element $^*C_{1,0}$ has the lowest column coordinate, this element is sent up to the MRN first. The same procedure is executed in a pipelined manner for the rest of the elements in the fiber until all the psums have been merged in a single fiber and sent to DRAM. In case the number of fibers in a row is greater than the number of multipliers (i.e., leaves in the tree), the controller needs to perform multiple passes to complete the final merge.

3.2.3 *Example of Gustavson's dataflow.*

Finally, for the same example matrices, Fig. 7 illustrates how Flexagon proceeds when the `Gust(M)` dataflow is selected. Similarly, the operation in this case proceeds in three well-differentiated phases.

Stationary phase: First, during the stationary phase, as many fibers of `A` (i.e., rows in matrix `A`) as possible are mapped spatially and sequentially in the multipliers. In the example, the multipliers, then keep two clusters, each in charge of calculating the psums for a different output row (i.e., rows 0 and 1 in the example).

Streaming phase: In the streaming phase, for each multiplier, the memory controller fetches and delivers the fiber of `B` (i.e., row of `B`) that corresponds to the column coordinate (i.e., k-iteration) associated to the mapped element of `A` in the multiplier. Every multiplier generates a partial output fiber which is merged with the rest of partial output fibers generated by the other multipliers allocated to the same fiber of `A`. An example of this generation is shown in Fig. 7. Here, we depict 6 streaming steps. The first multiplier keeps stationary the only one element in matrix `A` ($A_{0,1}$) so it receives the fiber of `B` indexed by column 1 (i.e., row 1). The second, third and fourth multipliers keep the elements $A_{1,0}$, $A_{1,2}$ and $A_{1,3}$, respectively, so they receive the fibers of `B` 0, 2 and 3, respectively. The first 3 steps show how the elements from the fibers of `B` are delivered cycle by cycle.

Merging phase: Similar to the `OP` dataflow, the merging phase combines both the accumulation and the merging operation, accumulating the elements (i.e., its values) in a certain node if their column coordinates match, or sending the element with the lowest column coordinate value. On the other hand, in `Gust` dataflow, we can merge the psums immediately after their generation, as a cluster of multipliers always generates fibers for the same row, but for different k iterations. When the number of elements in `A` fits into a cluster of multipliers, the output fiber generated by that cluster will be a final fiber, and the outputs can be sent directly to DRAM without being stored in the SRAM. Otherwise, when the number of elements in `A` exceeds the number of multipliers, the output fiber will be a partial fiber as multiple iterations are required, and therefore the fiber will require to be stored in the PSRAM, similar to what happens in the `OP` dataflow.

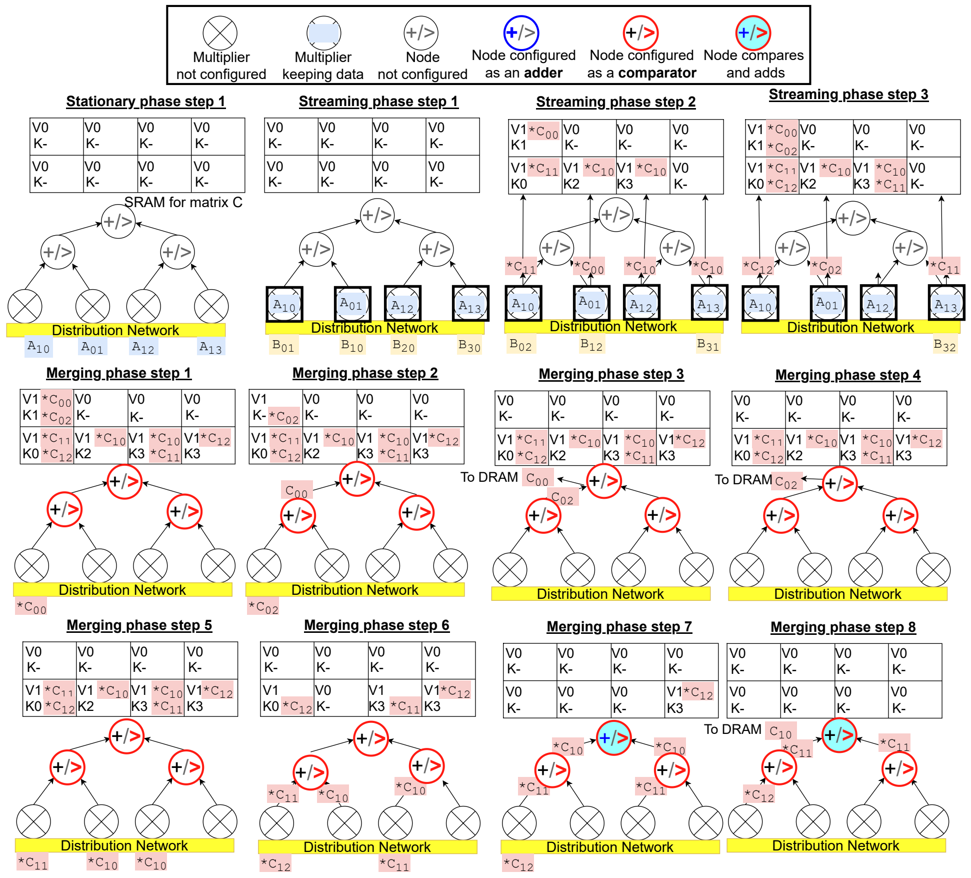


Figure 6: Example of Flexagon running SpMSpM using an `Outer-Product(M)` dataflow. "*" indicates that the outputs produced by the accelerator are psums and not final outputs. "*V*" in the PSRAM represents the valid bit and "*K*" indicates the *k* iteration tagged for a particular line.

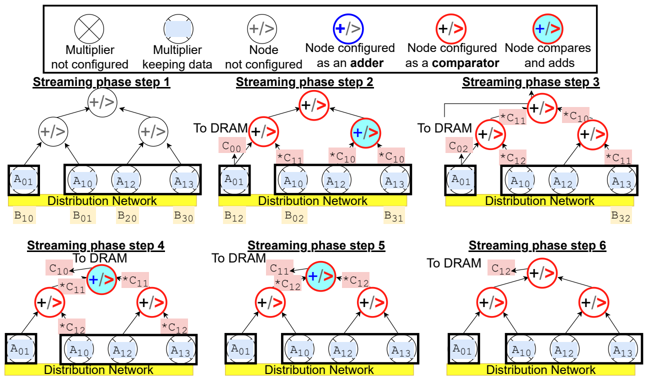


Figure 7: Example of Flexagon running SpMSpM using the `Gustavson(M)` dataflow. "*" indicates that the outputs produced by the accelerator are psums and not final outputs.

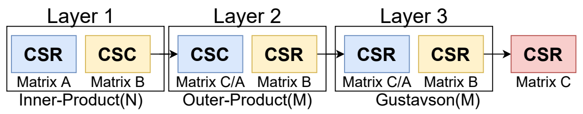


Figure 8: Example of three DNN layers being executed by running the combination `Inner-Product`, `Outer-Product` and `Gustavson` dataflows.

3.3 Combinations of inter-layer dataflows

As Table 3 shows, *M*-stationary dataflows output the elements in CSR format while *N*-stationary dataflows output the elements in CSC format. Flexagon supports the six dataflows and takes advantage of this observation to appropriately execute every possible sequence of DNN layers without requiring costly explicit hardware format conversions. This is the first work to support compressed outputs without explicit conversions. Fig. 8 shows an example of a DNN composed of three layers, demonstrating this feature. The first and the second layer are configured to execute inner and outer products respectively. Since the second layer needs activation in CSC, the first layer is Inner Product (N). The weights are assumed to be stored offline in both formats. The second layer produces the matrix in CSR format if it uses M-stationary. As a result, it could choose from inner product or Gustavson(M).

	IP(M)	*OP(M)*	*Gust(M)*	*IP(N)*	*OP(N)*	*Gust(N)*
IP(M)	✓	EC	✓	✓	EC	EC
OP(M)	✓	EC	✓	✓	EC	EC
Gust(M)	✓	EC	✓	✓	EC	EC
IP(N)	EC	✓	EC	EC	✓	✓
OP(N)	EC	✓	EC	EC	✓	✓
Gust(N)	EC	✓	EC	EC	✓	✓

Table 4: Possible dataflow transitions (EC stands for Explicit Conversion). Different rows represent the different outputs of the first layer and different columns represent the corresponding input to the second layer.

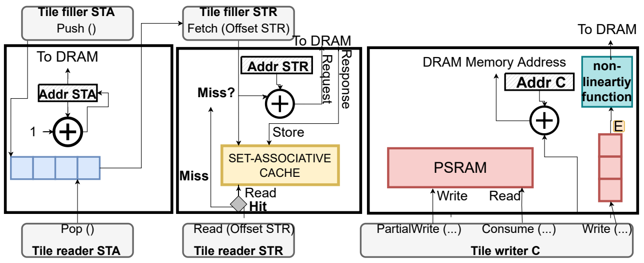


Figure 9: Memory structures in Flexagon.

Table 4 shows the transitions for each dataflow combination that do not require an explicit format conversion (green tick) and those that do (*Explicit Conversion* or EC). These combinations can be utilized by the mapper/compiler to generate the best sequence of dataflows that lead to the best performance and energy efficiency for a particular DNN execution.

3.4 Memory organization

In order to capture all dataflows, we have designed a customized L1 memory level specifically tailored for the common and different patterns among the three dataflows. Fig. 9 shows a schematic design for this L1 memory level. We use a separate memory structure and a different buffer idiom for data movement from/to each structure. To do so, every memory structure is operated by two controllers, the **tile filler** interfacing with the DRAM, and the **tile reader** interfacing with the datapath of the accelerator (i.e., the multipliers). Next, we describe each memory structure in detail:

Memory structure for the stationary matrix (FIFO): The elements of the stationary matrix are always read once and sequentially for the three dataflows, as they are kept stationary in the multipliers. To hide the access latency, we implement a 512-byte read-only FIFO. In order to save bandwidth and reduce the complexity: (1) the memory structure keeps the DRAM location of the stationary matrix in a register, so that the fibers are pushed implicitly into the FIFO, (2) we employ a single port to read and write.

Memory structure for the streaming matrix (Cache): The streaming matrix presents a more heterogeneous memory access pattern. In IP, every stationary phase (i.e., every iteration) causes the streaming of the entire matrix. In other words, there is a significant spatial locality and temporal locality every time the matrix is re-loaded. In the OP dataflow, the fibers of the streaming matrix are read just once and sequentially. In Gust dataflow, every fiber of the stationary matrix gathers *F* fibers of the streaming matrix, *F* being the number of non-zero elements in the fiber of the stationary matrix which are typically scattered all over the matrix, causing an irregular and unpredictable memory access pattern. To factor the worst-case Gust dataflow, we implement the memory structure for the streaming matrix as a traditional read-only set-associative cache. However, we implement this cache to operate on a virtual address space relative to the beginning of the streaming matrix, which lets us use shorter memory addresses and therefore save bandwidth and reduce tag lengths.

Memory structure for matrix C (PSRAM): To store the psums, we have designed a new buffer idiom called *PSRAM* , which is used for both OP and Gust dataflows. Fig. 6 shows the way this memory structure works, Fig. 9 shows a high-level diagram, and Fig. 10 delves into detail. The memory is organized into sets corresponding to different rows, and each set into lines for different K dimension within a row. Each line has a valid bit. Besides, we use a line tag to keep the column coordinate (i.e., the k-iteration) assigned to that line. Since the length of the output fiber is undetermined, it may occupy several (and non-consecutive) lines in the same row. This is essentially a way-combining scheme tagged by the k-iteration [31]. The tag is used by the row to locate whether a certain output fiber is still placed in the PSRAM. In order to read several fibers in parallel from the same set (i.e., to merge a particular row or column) we implement a multi-bank scheme organized across the lines within a set. Finally, we also include two fields to keep the byte location where the first and last elements are in the line.

PartialWrite(*row, k, E*): This operation is used to place an element in the PSRAM. The logic indexes the element by the *row* argument and then searches in parallel the line where the output fiber with the k identifier is being stored. If the output fiber exists (i.e., the k tags match), the PSRAM places the new element E into the last available position (indicated by the field *Last* in the metadata) of the last line. If the fiber does not exist, the logic searches the first available line and stores the element E in the first position of the line, enabling the valid bit and updating the K, *First* and *Last* fields in order to continue storing elements for the same K identifier in future accesses.

Consume(*Row, k*): The elements within a partial output fiber are placed in the PSRAM temporarily. They are read once to feed the accelerator and are no longer used again. This allows us to incorporate the **consume** operation, which reads and erases a particular element from the memory structure. In particular, the controller merges the partial output fibers row by row. To do so, the controller needs to read as many fibers as possible for the same row and for each fiber it uses the **consume** operation indicating the *row* and the fiber k to search. If there is an active line keeping the k fiber, the structure reads the next element from that fiber (indicated by the field *First*) and consumes it by increasing this field by one element. When the *First* and *Last* fields store the same value, the PSRAM detects that the line has been consumed and invalidates the line by setting the valid bit to 0.

Write(Offset, E): Apart from the PSRAM which is used to store partial output fibers, we also augment our memory structure with a FIFO which is used as a write buffer to hide the latency of sending out the final output fibers to DRAM. This structure implements the **Write** operation.

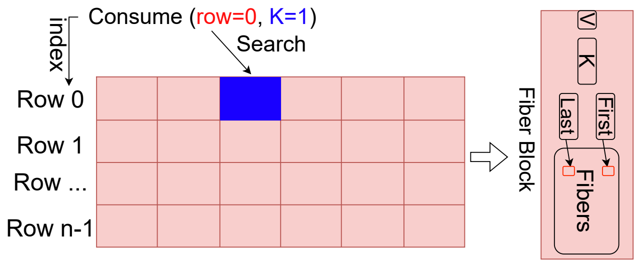
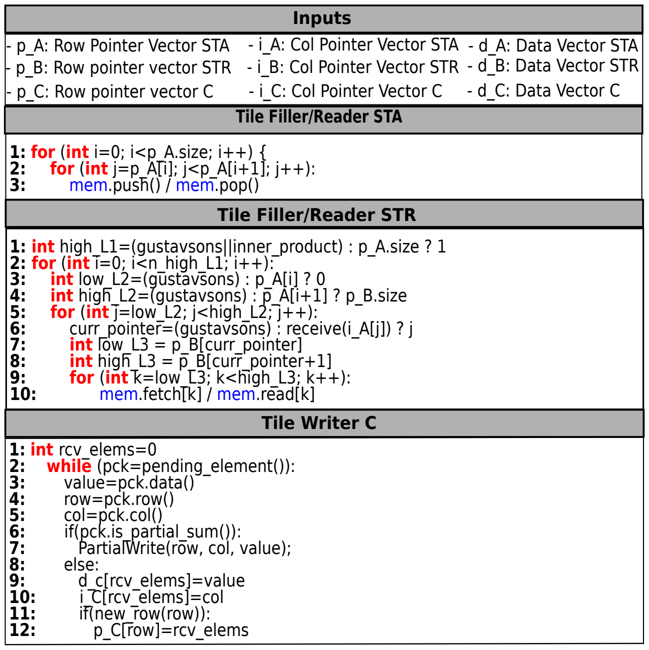


Figure 10: PSRAM overview.



Inputs

- p_A: Row Pointer Vector STA - i_A: Col Pointer Vector STA - d_A: Data Vector STA
- p_B: Row pointer vector STR - i_B: Col Pointer Vector STR - d_B: Data Vector STR
- p_C: Row pointer vector C - i_C: Col Pointer Vector C - d_C: Data Vector C

Tile Filler/Reader STA

```
for (int i=0; i<p_A.size; i++) {
    for (int j=p_A[i]; j<p_A[i+1]; j++):
        mem.push() / mem.pop()
```

Tile Filler/Reader STR

```
int high_L1=(gustavsons||inner_product) : p_A.size ? 1
for (int i=0; i<n_high_L1; i++):
    int low_L2=(gustavsons) : p_A[i] ? 0
    int high_L2=(gustavsons) : p_A[i+1] ? p_B.size
    for (int j=low_L2; j<high_L2; j++):
        curr_pointer=(gustavsons) : receive(i_A[j]) ? j
        int low_L3 = p_B[curr_pointer]
        int high_L3 = p_B[curr_pointer+1]
        for (int k=low_L3; k<high_L3; k++):
            mem.fetch[k] / mem.read[k]
```

Tile Writer C

```
int rcv_elems=0
  while (pck=pending_element()):
     value=pck.data()
     row=pck.row()
     col=pck.col()
     if(pck.is_partial_sum()):
        PartialWrite(row, col, value);
     else:
        d_c[rcv_elems]=value
        i_C[rcv_elems]=col
        if(new_row(row)):
           p_C[row]=rcv_elems
```

Figure 11: Pseudo-code of the tile filler STA, tile reader STA, tile filler STR, tile reader STR and the tile writer C. We fuse the fillers and readers in the same text box. STA: Stationary, STR: Streaming.

3.5 Memory controllers

Having one memory controller for each combination of dataflow and memory structure would be very costly in terms of area and power as it would require 30 logic modules to orchestrate the data (*6 dataflows* × *5 memory controllers*). In our design, we have unified the logic and each controller is able to be configured according to the memory access pattern of each dataflow. This way, as shown in Fig. 9, we only need two controllers to orchestrate the data for the memory structure which is kept stationary (i.e., the tile filler STA and the tile reader STA), two memory controllers to orchestrate the memory structure for the streaming matrix (i.e., the tile filler STR and the tile reader STR) and a single controller to orchestrate the memory structure for C (i.e., the tile writer C). Fig. 11 shows the code of these unified memory controllers.

Parameter	Value
Number of Multipliers	64
Number of Adders	63
Distribution bandwidth	16 elems/cycle
Reduction/Merging bandwidth	16 elems/cycle
Total Word Size (Value+Coordinate)	32 bits
L1 Access Latency	1 cycle
L1 STA FIFO Size	256 bytes
L1 STR cache Size	1MiB
L1 STR Cache Line Size	128 bytes
L1 STR Cache Associativity	16
L1 STR Cache Number of Banks	16
PSRAM	256 KiB
DRAM size	16 GiB
DRAM access time / Bandwidth	100 ns / 256 GB/s

Table 5: Configuration parameters of Flexagon.

4 EXPERIMENTAL METHODOLOGY

Simulation Infrastructure: For a detailed evaluation of Flexagon, we have implemented a cycle-level microarchitectural simulator of all on-chip components of our accelerator by leveraging the STONNE framework [20]. To faithfully model the whole memory hierarchy including an HBM 2.0 off-chip DRAM, we have connected the simulated accelerator to the Structural Simulation Toolkit [18]. Table 5 shows the main parameters of the architecture we have configured for the rest of the evaluation. We compare our results against three state-of-the-art accelerators: SIGMA-like as an example of an `IP` accelerator, SpArch-like as an example of an `OP` accelerator and GAMMA-like as an example of a `Gust` accelerator.

We use the term *-like* in GAMMA-*like*, SIGMA-*like* and SpArch-*like* to reflect the fact that we capture their most relevant characteristics (i.e., their essence) in our simulator in a fair and normalized fashion. Specifically, we focus on the dataflow, which is a critical part for the efficiency of the accelerator; the DN, MN and RN components, which define the accelerator size and bandwidth; and the on-chip memory structures, which determine the capacity of the accelerator to store data close to the processing elements. We note that these are given extra on-chip area as appropriate. The main sources of efficiency in SIGMA, SpArch and GAMMA are the FAN reduction network, merge network and the fiber cache respectively rather than a specifically engineered design-point. Thus, we believe that the comparison against the key features of these designs captured by SIGMA-like, SpArch-like and GAMMA-like is justifiable, since our aim is to establish the advantages of flexibility and our ability to achieve it without major area overhead rather than obtain a specifically engineered design point.

For the three accelerators, we model the same parameters presented in Table 5, and we only change the memory controllers to deliver the data in the proper order according to its dataflow. We also compare Flexagon against the implementation from Intel MKL [32] running on a 4-core 8-thread Intel(R) Core(TM) i5-7400 CPU @ 3.00 GHz. Each core implements a 128 KiB L1 cache, a 1 MiB L2 cache and a shared 6 MiB L3 cache. We do not include GPU results because existing GPU SpMSpM implementations do not support sparse weights+activations natively [33, 36], thus performing similarly to CPU MKL as reported in [34, 35].

To demonstrate the benefits of Flexagon, our evaluation methodology considers the following three different angles:

Layer	M, N, K	spA	spB	csA	csB	csC
SQ5	64, 2916, 16	68	11	1.2	162	728
SQ11	128, 729, 32	70	10	4.8	82	364
R4	256, 3136, 64	88	9	7.6	709	3136
R6	64, 2916, 576	89	53	16	3086	728
S-R3	64, 5329, 576	89	46	16	6422	1332
V0	128, 12100, 576	90	61	29	21357	12321
MB215	128, 8, 512	50	0	128	16	4
V7	512, 144, 4608	90	94	921	177	288
A2	384, 121, 1728	70	54	777	373	181

Table 6: Representative DNN layers selected for the evaluation. sp{A,B}=sparsity of matrix {A,B} (in %), cs{A,B,C}=compressed size of matrix {A,B,C} (in KiB).

	SIGMA-like	SpArch-like	GAMMA-like	Flexagon
DN	Tree	Tree	Tree	Tree
MN	Linear	Linear	Linear	Linear
RN	FAN	MergerS	MergerG	MRN

Table 7: Main building blocks to model the SIGMA-like, SpArch-like, GAMMA-like and Flexagon accelerators. DN=Distribution Network, RN=Reduction Network and MN= Multiplier Network.

End-to-End Evaluation: To truly prove the performance benefits of Flexagon, we have carried out end-to-end execution of complete DNN models (see Table 2) in our simulated accelerators. These models are present in the MLPerf benchmark suite [29] and we take other models for completeness. As it may be appreciated, we consider very diverse DNN models in terms of the number of layers and sizes. The matrices involved in the execution of each DNN layer range from 0.003 MiB up to 63.41 MiB (see average compressed sizes in Table 2), thereby our evaluation is comprehensive as there are many situations where matrices cannot completely fit on chip (Flexagon uses a total of 1 MiB SRAM memory for storing input matrices).

Layer-wise evaluation: Since explaining the results requires delving into a finer-grained detail, we have selected 9 representative layers extracted from the execution of the DNN models. Table 6 shows these layers together with the characteristics of each layer.

RTL results: We implemented the main building blocks (i.e., the DN, MN, RN and the on-chip memory) of the accelerators considered in this work (shown in Table 7). For an apples-to-apples comparison of overheads, the four architectures use the same tree topology for the DN, the same linear array of multipliers for the MN and vary the RN. For the SIGMA-like architecture, we utilize the FAN network [28] as the RN for flexible-sized reductions. For the SpArch-like and GAMMA-like architectures, we use a merger [23, 35] to merge the partial sums produced after the multiplications. Finally, for Flexagon we utilize the unified MRN explained in Section 3.

For synthesis, we use MAERI BSV [1] to generate the 64-MS distribution network and the multiplier network. In addition, we have implemented in RTL a 64-wide merger and our MRN. We use Synopsys Design Compiler and Cadence Innovus Implementation System for synthesis and place-and-route, respectively, using TSMC 28nm GP standard LVT library at 800 MHz. To obtain the area and power numbers of the memory structures, we have used CACTI 7.0 [11] for the same technology node and frequency.

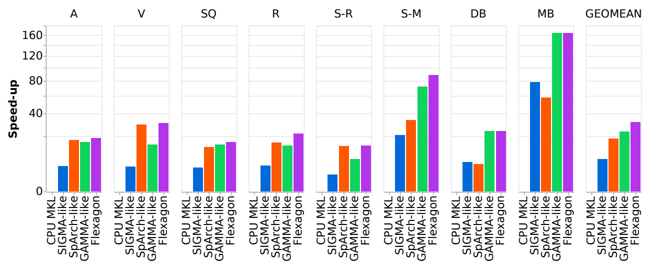


Figure 12: Performance comparison between CPU MKL, SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across the 8 DNN models (speed-up against SIGMA-like).

5 RESULTS

5.1 End-to-end results

Figure 12 compares the performance obtained with the CPU MKL, the three contemporary fixed-dataflow accelerators (SIGMA-like, SpArch-like and GAMMA-like) and with Flexagon when running the 8 DNN models (speed-ups with respect to the results obtained with the CPU MKL). The total numbers of cycles for CPU MKL are reported in the last column of Table 2.

The first observation is that there is no fixed-dataflow accelerator that can obtain the highest performance for all the 8 DNN models. In particular, for *Alexnet* (A), *VGG-16* (V), *Resnets-50* (R) and *SSD-Resnets* (S-R) the SpArch-like accelerator is 5.26× and 1.49× on average faster than the SIGMA-like and GAMMA-like architectures, respectively. Conversely, for *Squeezenet* (SQ), *SSD-Mobilenets* (SM), *DistilBert* (DB) and *MobileBert* (MB), the GAMMA-like accelerator obtains the best performance (average improvements of 3.28× and 2.41× against the SIGMA-like and SpArch-like, respectively).

The second and most noteworthy observation is that Flexagon can outperform the other three fixed-dataflow accelerators in all cases, attaining average speed-ups of 4.59× (vs. SIGMA-like), 1.71× (vs. SpArch-like) and 1.35× (vs. GAMMA-like). This is due to the combination of its flexible interconnects, explicitly decoupled memory structures and unified memory controllers that enable using the most efficient dataflow for each layer.

Finally, we observe that Flexagon significantly outperforms the CPU MKL as the hardware is specifically designed to perform the SpMSpM operation. Overall, we find that Flexagon obtains a speed-up of 31× on average (benefits from 13× up to 163× are observed).

5.2 Layer-wise results

Detailing the reasons behind the benefit observed for some DNN models for a particular dataflow requires a deeper delve into every DNN layer execution. To make the study feasible (we run over a hundreds of layers), next, we present a comprehensive study for a selected set of nine representative DNN layers (Table 6). These layers are chosen according to the dataflow from which they benefit the most –The first three layers in the table benefit from IP (*SQ5*, *SQ11* and *R4*), the second ones from OP (*R6*, *S-R3* and *V0*), and the third ones from Gust (*MB215*, *V7* and *A2*).

Figure 13 shows a performance comparison running these selected layers using our simulated accelerators (again, speed-ups are computed with respect to SIGMA-like). As expected, as shown in

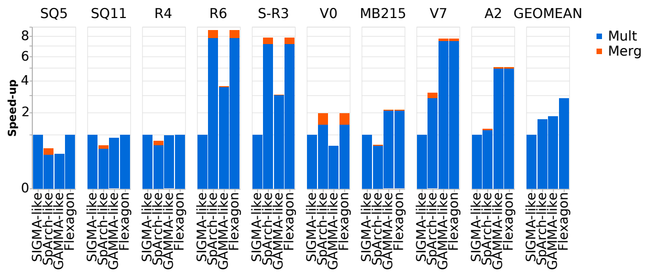


Figure 13: Performance comparison between SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across our 9 DNN layers (speed-up against the SIGMA-like one).

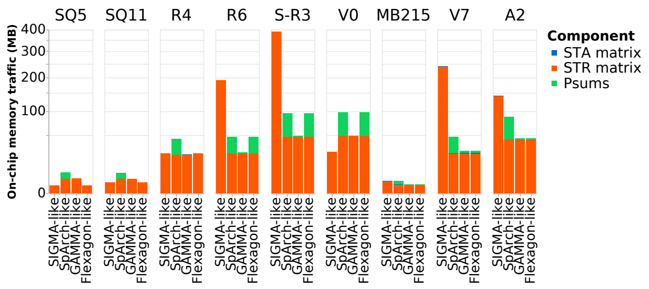


Figure 14: Memory traffic (MB) that flows through the on-chip memory hierarchy for SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across our 9 DNN layers.

the figure, in the case of the first group of IP-friendly layers, the SIGMA-like architecture obtains average speed-ups of 1.53× and 1.40× against the SpArch-like and the GAMMA-like architectures, respectively. The next three OP-friendly layers (i.e., *R6*, *S-R3* and *V0*), the SpArch-like architecture obtains an average increased performance of 5.07× and 2.66× against the SIGMA-like and GAMMA-like architectures. Finally, for the last three Gust-friendly layers, the best performance is obtained by the GAMMA-like architecture, experimenting 4.37× and 3.19× faster executions than the SIGMA-like and the SpArch-like architectures, respectively. More remarkable is that Flexagon beats all of them, always reaching the performance of the best case. Overall, by properly configuring the control logic of Flexagon according to the most suitable dataflow for each layer, our accelerator is able to attain 2.81×, 1.69×, and 1.55× speed-ups against the SIGMA-like, SpArch-like and GAMMA-like accelerators.

Figures 14, 15 and 16 help us understand these results. Specifically, Figure 14 shows the amount of on-chip memory traffic (expressed in MBs) that relays between our on-chip memory hierarchy (i.e., the reads from the STA FIFO and from the STR cache and the reads/writes from/to the PSRAM) and the distribution network after running the SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across our nine DNN layers. Figure 15 plots the cache miss rate of the STR cache after running the layers, and Figure 16 shows the amount of off-chip traffic (expressed in KBs) that in consequence, flows between this STR cache and the DRAM.

The first observation that we would like to make from Figure 14 is the negligible traffic that is fetched from the memory structure for the STA matrix (inappreciable fractions of the bars in blue color). This is basically due to the fact that the stationary data is kept stationary in the multipliers once it is read for the rest of the execution, as it is explained in Section 3. For this reason, this memory structure does not have a significant impact on the final performance of the executions regardless of the dataflow that is configured. In contrast, the amount of traffic required to fill the structure for the STR matrix and the PSRAM heavily varies layer by layer and across dataflows (fractions of the bars in orange and green colors respectively), hence determining the final performance of the layer execution.

As we can appreciate, for every layer execution, both the SpArch-like and GAMMA-like architectures present the same amount of traffic for the memory structure for the STR matrix. This is so because the elements that are read are exactly the same during the multiplying phase. On the contrary, the traffic generated for the STR matrix in the SIGMA-like architecture may vary based on the intersections that exist between the elements that are mapped in the multipliers and the elements in the streaming matrix. More specifically, the matrices B of the layers that benefit from the SIGMA-like architecture (i.e., *SQ5*, *SQ11* and *R4*) are relatively small (up to 709 KB) and present a low sparsity ratio (average of 10.1%) which leads to most of the values from matrix A to intersect and therefore generate the same amount of traffic as the SpArch-like and the GAMMA-like accelerators.

Since the IP dataflow does not require merging the partial sums as they are internally accumulated (observe the number of partial sums sent to the PSRAM for the SIGMA-like architecture is always 0) this dataflow obtains the best performance. An outlier for this behaviour is observed for the *V0* layer. Here, the traffic generated for the STR matrix in the SIGMA-like architecture is lower than the traffic generated in the SpArch-like and GAMMA-like architectures. However, this workload experiences higher runtime. The reason for this is the large size of the matrix B (21.3 MiB) which causes that it has to be reloaded several times, experimenting an L1 miss rate of 3.13% (see Figure 15), significantly higher than the L1 miss rates obtained for the SpArch-like and GAMMA-like architectures (i.e., 0.36% and 2.30%) which translates into increased off-chip memory traffic (see Figure 16). This higher traffic provokes that the multiplying phase takes longer for the SIGMA-like architecture than for both the multiplying and merging phase for the SpArch-like architecture. When the number of intersections is low, the SIGMA-like architecture experiments higher number of cycles overheads due to this architecture accesses to many more data elements. This is also observed in the six layers that do not benefit from the SIGMA-like architecture (i.e., *R6*, *S-R3*, *V0*, *MB215*, *V7* and *A2*), experiencing on average 5.68× and 2.27× higher on-chip traffic than the SpArch-like and GAMMA-like architectures.

On the other hand, out of these six layers, the main difference in performance that defines them comes from the size of matrix B. The second group of layers (i.e., *R6*, *S-R3* and *V0*) that benefit from the SpArch-like architecture have a large size for matrix B (see Table 6). This implies that the GAMMA-like architecture cannot fit the rows of B entirely in the memory structure for the STR matrix, causing higher L1 miss rates. Observed average L1 miss rate (see Figure 15) experimented in the execution of these three layers is 0.39% for the SpArch-like architecture and 2.43% for the

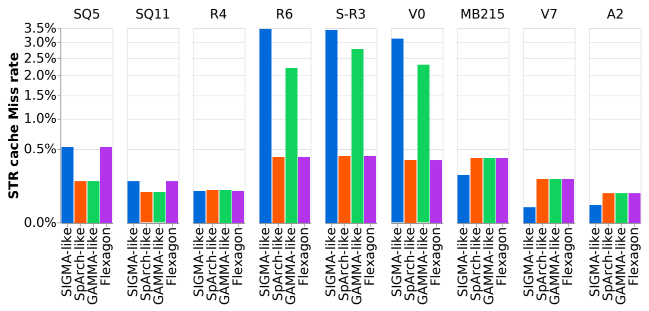


Figure 15: STR cache miss rate for the SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across 9 DNN layers.

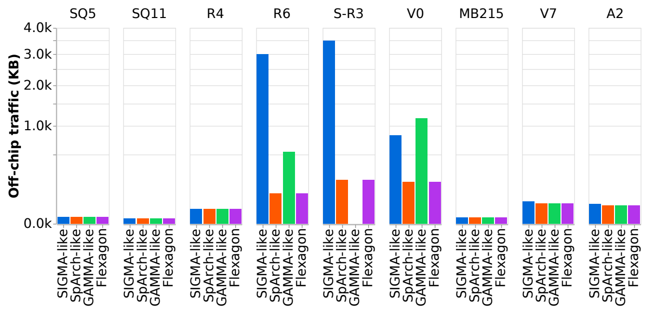


Figure 16: Off-chip data traffic for the SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across 9 DNN layers.

GAMMA-like architecture. This translates into 6.25× more traffic for GAMMA which causes the degradation in performance.

In the last group of layers (i.e., *MB215*, *V7* and *A2*) the sizes of matrices B are much smaller (up to 373KB as observed in Table 6) and therefore both SpArch-like and GAMMA-like architectures experience the same L1 miss rates and off-chip data traffic. In this scenario, the GAMMA-like architecture is more efficient as it is able to compute the reduction phase and the merging phase at the same time –Observe the orange bar for the GAMMA-like cases in the Figure 13 is not significant as the merge phase is computed in parallel within the multiplying phase (i.e., blue bar).

5.3 RTL results

Table 8 shows a breakdown of the total amount of area (mm^2) and power (mW) obtained for the 64-MS SIGMA-like, SpArch-like, GAMMA-like and Flexagon accelerators. For each case, we show the results for the main architectural components: Distribution Network (DN), Multiplier Network (MN), Reduction/Merger Network (RN), the cache structure for the streaming matrix (Cache) and the PSRAM .

In terms of area, we observe that Flexagon introduces an overhead of 25%, 3% and 14% with respect to the area of the SIGMA-like, SpArch-like and GAMMA-like accelerators, respectively. As we can see, the area of the four accelerators is mostly dominated by the memory structures. Specifically, we observe that the cache for the streaming matrix represents 93%, 76%, 85% and 74% of the total amount of area for the SIGMA-like, SpArch-like, GAMMA-like

Component	SIGMA-like	SpArch-like	GAMMA-like	Flexagon
Area Results				
DN (mm^2)	0.04	0.04	0.04	0.04
MN (mm^2)	0.07	0.07	0.07	0.07
RN (mm^2)	0.17	0.07	0.07	0.21
Cache (mm^2)	3.93	3.93	3.93	3.93
PSRAM (mm^2)	-	1.03	0.51	1.03
Total (mm^2)	4.21	5.14	4.62	5.28
Power Results				
DN (mW)	2.18	2.18	2.18	2.18
MN (mW)	3.29	3.29	3.29	3.29
RN (mW)	248	64.48	64.48	312
Cache (mW)	2142	2142	2142	2142
PSRAM (mW)	-	538	269	538
Total (mW)	2396	2750	2481	2998

Table 8: Post-layout area and power obtained for SIGMA-like SpArch-like, GAMMA-like and Flexagon accelerators.

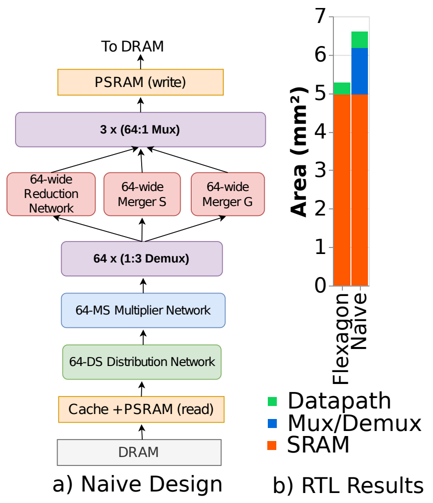


Figure 17: a) High-level overview of a non-unified naive design. b) Area comparison between Flexagon and the naive design.

and Flexagon architectures, respectively. Besides, the area of the PSRAM represents 20%, 11% and 19% with respect to the SpArch-like, GAMMA-like and Flexagon accelerators, respectively. Since the SIGMA-like architecture employs an IP dataflow, this accelerator does not need this structure, which explains the reason for having the lowest area. Also, the area of the PSRAM in the GAMMA-like accelerator is half the area in the SpArch-like and Flexagon accelerators as it requires storing fewer partial sums, which explains the area reduction. Obviously, Flexagon needs support for the worst-case OP dataflow and needs the highest PSRAM overhead. Finally, note that our MRN is 28% and 128% larger than the area of the FAN and the merger, but this does not translate into high overall overhead as the MRN takes only 4% out of the total area for Flexagon.

Figure 17 proves the area benefits of unifying the RN and the merger into a single network (MRN). To do so, we have sketched a 64-MS naive accelerator design similar to Flexagon, but utilizing separate networks for each dataflow (see Figure 17a). We use the term *naive* here to emphasize the fact that the design simply replicates the reduction network 3 times (one per dataflow). As it may be seen, the reduction and merger networks share the same multiplier and distribution networks as well as the same SRAM capacity. The design requires extra links, muxes and demuxes to connect the pieces. At the bottom side, the MN connects to three different networks, and therefore, requires 64 (1:3) demultiplexers. At the top side, each node from the merger and reduction network has to be connected to memory requiring 3 costly (64:1) multiplexers and connections. Figure 17b shows the inefficiencies of this naive design. As we can see, the three separate networks (i.e., RNs and mergers) introduce an area overhead of just 2% as the designs are dominated by the SRAM area (e.g., 74% of area for Flexagon). The significant area penalty introduced by the naive design comes from the extra multiplexers, demultiplexers and corresponding connections, introducing an area overhead of 25% over Flexagon. Note that in larger configurations (i.e., greater number of multipliers) this area overhead would even increase.

In terms of power, we observe the same trends. We find that the Flexagon accelerator consumes 25%, 9% and 21% more power than the SIGMA-like, SpArch-like and GAMMA-like accelerators. The slightly higher overhead of Flexagon against the aforementioned area results comes mostly from the MRN as this module represents a larger fraction of total consumption (10%, 2.34%, 2.60% and 10.41% out of the SIGMA-like, SpArch-like, GAMMA-like and Flexagon accelerators are observed, respectively). This, together with the fact that the MRN consumes 25% and 284% more than the FAN RN and the merger, explains the results. In spite of the overhead introduced, in Figure 18 we illustrate that Flexagon is still more performance/area efficient. Specifically, we consider both achieved speed-ups and area requirements of each design. The area requirements are normalized with respect to the SIGMA-like case, which is also the reference for the calculation of the speed-ups. Note that the NLP models like *MobileBert* (*MB*) and *DistilBert* (*DB*) achieve a better efficiency with the GAMMA-like accelerator. Nevertheless, this is due to as explained before, most of the layers (84% in DistilBert (*DB*) and 100% in MobileBert (*MB*)) for these models work better with the `Gustavson` dataflow, making the area overhead introduced by the Flexagon accelerator unnecessary. Consequently, we can clearly see that, overall, Flexagon reaches the best compromise between performance and area consumption (the higher Speed-up/Area values). In comparison, we find that, on average, our accelerator obtains 18%, 67% and 265% better performance/area efficiency across the execution of our 8 DNN models with respect to the GAMMA-like, SpArch-like and SIGMA-like accelerators. This makes Flexagon the best candidate for running heterogeneous sparse DNN workloads.

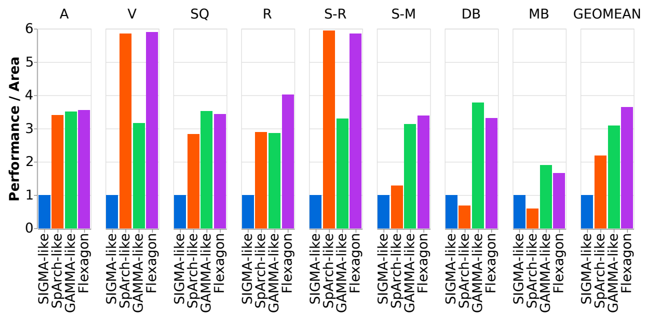


Figure 18: Performance/Area obtained after running the SIGMA-like, SpArch-like, GAMMA-like and Flexagon architectures across our 8 DNN models.

6 RELATED WORK

Sparse DNN Accelerators: Sparse matrix multiplications have been prime targets of acceleration for AI and HPC workloads. Several sparse DNN accelerators have been proposed for SpMM, SpGEMM and Sparse convolution [2, 6, 8, 10, 13, 17, 23, 25, 28, 30]. These accelerators have support for sparse execution via compression of one or both operands into formats like CSR, CSC, bitmap, CSF, etc. This reduces the memory footprint and the number of multiplications. As Table 1 shows, prior sparse accelerators have picked either one of `IP`, `OP` and `Gust`(row-wise product) dataflows. We show that flexibility to support multiple dataflows is beneficial for performance and performance per area.

Frameworks for flexible accelerators: Prior works in the direction of flexibility include hardware widgets and design-space exploration tools for CGRAs. MINT [27] is a format converter widget that supports multiple sparse formats. Prior works Garg et al. [5], coSPARSE [4] and SparseAdapt [22] propose frameworks for efficient sparse execution on CGRAs. However, to the best of our knowledge, this is the first work that proposes an accelerator for Sparse DNNs which exploits all three dataflows.

7 CONCLUSION

This work proposes Flexagon, the first SpMSpM accelerator design that offers `IP`, `OP` and `Gust` dataflows on a homogeneous hardware substrate. Flexagon revolves around a novel tree-based network (MRN) that supports both reduction of dot products and the merging of partial sums, and a special L1 on-chip memory organization, specifically tailored to the different access characteristics of the input and output compressed matrices. By using the dataflow that best matches the characteristics of each DNN layer, we show that Flexagon brings significant improvements in performance/area efficiency over SOTA fixed-dataflow sparse accelerators.

ACKNOWLEDGMENTS

This work was supported by grants TED2021-130233B-C33 and RYC2021-031966-I both funded by MCIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR. F. Muñoz-Martínez was supported by grant 20749/FPI/18 from Fundación Séneca. A part of the work was supported by the ARIAA co-design center funded by the U.S. Department of Energy (DOE) Office of Science, Advanced Scientific Computing Research program.

REFERENCES

[1] [n. d.]. MAERI code v1. https://github.com/hyoukjun/MAERI.
[2] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (June 2019), 292 – 308.
[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv: 1810.04805v2 (2019)* (May 2019).
[4] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong-hyeon Park, Magnus Morton, Trevor Mudge, Murray Cole, Michael O'Boyle, Chaitali Chakrabarti, and Ronald Dreslinski. 2021. CoSPARSE: A Software and Hardware

Reconfigurable SpMV Framework for Graph Analytics. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 949–954. https://doi.org/10.1109/DAC18074.2021.9586114

[5] Raveesh Garg, Eric Qin, Francisco Muñoz-Martínez, Robert Guirado, Akshay Jain, Sergi Abadal, José L Abellán, Manuel E Acacio, Eduard Alarcón, Sivasankaran Rajamanickam, and Tushar Krishna. 2022. Understanding the Design-Space of Sparse/Dense Multiphase GNN dataflows on Spatial Accelerators. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[6] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 151–165.

[7] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (sep 1978), 250–269.

[8] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[9] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv: 1510.00149v5 (2016)* (Feb. 2016).

[10] 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 Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[11] HP Laboratories. [n. d.]. CACTI 7.0: A Tool to Model Caches/Memories, 3D stacking, and off-chip IO. https://github.com/HewlettPackard/cacti.

[12] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Proceedings of the 44th Annual International Symposium on Computer Architecture* 45, 2 (jun 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[13] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 600–614.

[14] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. taco: a tool to generate tensor algebra kernels. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Oct. 2017), 943–948.

[15] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 754–768. https://doi.org/10.1145/3352460.3358252

[16] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (March 2018).

[17] Ching-En Lee, Yakun Sophia Shao, Jie-Fang Zhang, Angshuman Parashar, Joel Emer, Stephen W Keckler, and Zhengya Zhang. 2018. Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks. In *SysML Conference*, Vol. 120.

[18] Janssen Curtis Lee, Helgi Adalsteinsson, Scott Cranford, Joseph P. Kenny, Ali Pinar, David A. Evensky, and Jackson R. Mayo. 2010. A Simulator for Large-Scale Parallel Computer Architectures. *IJDST vol.1, no.2* (2010), 57–73.

[19] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. 2019. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500* (2019).

[20] Francisco Muñoz-Martínez, 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)*. 201–213. https://doi.org/10.1109/IISWC53511.2021.00028

[21] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[22] Subhankar Pal, Aporva Amarnath, Siying Feng, Michael O'Boyle, Ronald Dreslinski, and Christophe Dubach. 2021. SparseAdapt: Runtime Control for Sparse Linear Algebra on a Reconfigurable Accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1005–1021. https://doi.org/10.1145/3466752.3480134

[23] 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 *ISCA*.

[24] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315. https://doi.org/10.1109/ISPASS.2019.00042

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

[26] 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 *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1014–1024.

[27] 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 *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1014–1024. https://doi.org/10.1109/IPDPS49936.2021.00110

[28] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet 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 *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[29] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 446–459.

[30] 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)*. IEEE, 766–780.

[31] Rubén Titos-Gil, Antonio Flores, Ricardo Fernández-Pascual, Alberto Ros, Salvador Petit, Julio Sahuquillo, and Manuel E. Acacio. 2019. Way Combination for an Adaptive and Scalable Coherence Directory. *IEEE Transactions on Parallel and Distributed Systems* 30, 11 (2019), 2608–2623. https://doi.org/10.1109/TPDS.2019.2917185

[32] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel Math Kernel Library. *High-Performance Computing on the Intel Xeon Phi* (June 2014).

[33] Ziheng Wang. 2020. SparseRT: Accelerating Unstructured Sparsity on GPUs for Deep Learning Inference. *arXiv preprint arXiv: 2008.11849v1 (2020)* (Aug. 2020).

[34] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. 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*. 687–701.

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

[36] Maohua Zhu and Yuan Xie. 2010. Taming Unstructured Sparsity on GPUs via Latency-Aware Optimization. *2020 57th ACM/IEEE Design Automation Conference (DAC)* (Oct. 2010).

Received 2022-10-20; accepted 2023-01-19