Hardware Accelerators for Graph Convolutional Networks

Kareem Ahmad

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-148
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-148.html

May 21, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Hardware Accelerators for Graph Convolutional Networks

by Kareem Ahmad

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Sophia Shao
Research Advisor

5 | 21 | 2021

(Date)

* * * * * * *

Professor Borivoje Nikolic
Second Reader

5/21/2021

(Date)

Hardware Accelerators for Graph Convolutional Networks

Kareem Ahmad

(Dated: May 21, 2021)

Most datasets in real-world systems have relationships that are not Euclidean in nature, and are instead best described using graphs. The development of Graph Convolutional Networks (GCNs) has proven to be an efficient approach to learning on graph-structured data. Due to the sparse nature of graphs, however, traditional systolic-array based matrix-algebra accelerators do not achieve high levels of utilization when running inference on GCNs. In this paper, we characterize the performance of GCNs in terms of its four major operations: dense direct memory access (dDMA), sparse direct memory access (sDMA), dense-dense matrix multiplication (GeMM), and sparse-dense matrix multiplication (SpMM), laying the groundwork for adding efficient GCN support to Gemmini, a configurable systolic-array based GeMM accelerator. We also propose the addition of a sparse-to-dense decompression DMA Engine to Gemmini, providing a reference implementation in Spike—the RISC-V ISA-level simulator—and C tests.

I. INTRODUCTION

Many real-world datasets have relationships that are best described by a graph. Just as 2D convolutions take advantage of the relationship between adjacent pixels to learn efficiently, efficient learning on graph-structured datasets requires taking advantage of the relationships between nodes in the dataset. This is more intuitive in the case of a citation network. A citation network, like the citeseer or cora datasets, is a graph where each node is an article and edges represent citations from one paper to another. Each article, or node, has a vector of features, typically produce by a bag-of-words encoding. Citation networks are often used to train GCNs to classify articles by subject category, and it is here that the benefits of a GCN over DNNs or other neural networks becomes apparent. A DNN trained on a citation network, will learn how to classify articles based solely on their features, but a GCN—which fundamentally takes into account the connections between nodes—will learn to classify articles not only based on their own content, but also based on the the content of the papers they cite. In short, by using the edge weights of a graph as a notion of proximity and generalizing the idea of convolutions to graphs, we enable efficient learning on graph-structured datasets. In this manner, GCNs enable solving a vast array of problems including article classification in citation networks [3], author recognition [9, 4], rating prediction for recommendation systems [8], graph classification [4], and more.

In this thesis, we begin with an overview of graph-convolutions in Section II, motivating the choice of forward propagation rules, and examining the general structure of a GCN. Next, in Section III, we provide an overview of the current state of hardware accelerators for GCNs, before delving into the theoretical performance of a GCN on systolic-array based accelerators in Section IV. In Section V, we propose a new sparse-to-dense DMA decompression engine for Gemmini, with a software interface and spike implementation. We provide baseline performance numbers for GCNs in the current Gemmini as well as predicted performance with sDMA and SpMM implementations in Section VI. Finally, in Section VII we provide closing remarks and discuss the future work in this research.

II. GRAPH CONVOLUTIONS

The problem when trying to construct a graph convolution by mapping a 2D convolution onto a graph, is that unlike an image where the structure around any given pixel is regular, the structure around a node can be arbitrarily complex. For example, there no limit on the degree of a node in a graph, nor are there notions of direction associated with an edge. As such, all edges are more or less equal except for their weights when available. In addition, unlike a traditional 2D convolution where inference is done on the whole, a graph convolution must preserve structure of a graph, lest information about the original nodes be lost. As a result, we need to reconsider what a convolution is fundamentally trying to do.

In the 2D case, a convolution is a way of collecting information from adjacent pixels into a single unit—a single pixel becomes a weighted sum of the adjacent pixels. Thus, defining a convolution on a graph requires a notion of proximity between nodes. The most natural choice here is an adjacency matrix, so we represent our graphs with an adjacency matrix, $A \in \mathbb{R}^{N\times N}$, and a feature matrix, $F^{(i)} \in \mathbb{R}^{N\times H^{(i)}}$. Since we must preserve the structure of the graph, the weights of the convolution can only act on and change the number of features associated with a node. To this end, we represent the weights in a graph convolution as a matrix $W^{(i)} \in \mathbb{R}^{H^{(i)}\times H^{(i+1)}}$, where $H^{(i)}$ is the number of input features and $H^{(i+1)}$ is the number of output features. Since we must group information from adjacent nodes before applying the weight matrix, we may consider summing over the features of neighboring nodes by premultiplying features with the adjacency matrix:

$$F^{(i+1)} = AF^{(i)}W^{(i)}. \tag{1}$$

While this convolution does account for the graph

structure, it places more weight on nodes of a higher degree. We can adjust for this by changing our sum to an average; that is, by premultiplying the adjacency with the inverse of the degree matrix.

$$F^{(i+1)} = D^{-1} A F^{(i)} W^{(i)}. \tag{2}$$

This better expresses the idea of a graph convolution, but has the flaw of not considering a node's own features. This can be remedied by adding the identity matrix to the normalized adjacency matrix:

$$F^{(i+1)} = \left(I_N + D^{-1} A\right) F^{(i)} W^{(i)}. \tag{3}$$

This definition can be further improved by using the "renormalization trick" proposed by Kipf et. al. [3]:

$$F^{(i+1)} = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} F^{(i)} W^{(i)}, \tag{4}$$

where $\tilde{A}$ is the adjacency matrix including self-loops, and $\tilde{D}$ is the degree matrix including self-loops. This is the convolution we use moving forward, and can be simplified into two operations if we precompute $\hat{A} = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$.

$$F^{(i+1)} = \hat{A} F^{(i)} W^{(i)}. \tag{5}$$

A GCN, then, is simply a network built up of graph-convolution layers and nonlinear activations. In a typical GCN, $\hat{A}$ is sparse, while both F and W are dense. As a result, each graph convolution becomes a SpMM followed by a GeMM. Since the number of input features is typically larger than the number of output features, $\hat{A}(FW)$ will usually be the more efficient ordering. It should be noted, however, that there are times when F is sparse. The input features to a GCN, for example, are often ultra-sparse (have density less than 1%), such as the bag-of-words embedding of articles in a citation network. In some cases, a ReLU activation layer can also cause the output features of a layer to become sparse, as negative values get clamped to zero. These cases however, have more moderate sparsity on the order of 50%. In these cases, the $\hat{A}(FW)$ ordering is still preferable, and both matrix multiplications become SpMM. Another object of note is that while the renormalized adjacency matrix, $\hat{A}$ is very sparse, it's sparsity typically follows a power-law distribution [2]. That is to say that the distribution of rows with x non-zero elements is proportional to $x^{-\beta}$ for a constant $\beta > 0$. This fact usually requires some sort of load balancing in hardware accelerators targeting GCNs. This distribution for the first 1000 entries of the citeseer citation-network dataset is shown in Figure 1.

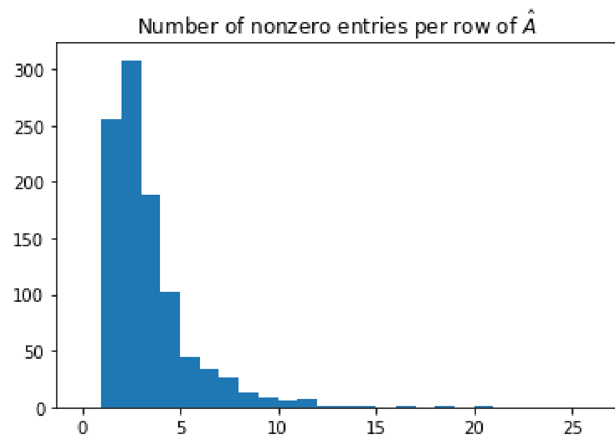

FIG. 1. Power law distribution of non-zeros in the renormalized adjacency matrix for the first 1000 entrees of the citeseer dataset.

III. BACKGROUND

A number of accelerators have been developed targeting SpMM [2, 10, 5, 7], and while their architectures may seem quite varied, they tend to share a similar high-level organization as shown in fig. 2. The core of each of these accelerators is an array of Processing Engines (PEs), that is fed from local memory through a local DMA engine. This local DMA engine often doubles as, or works closely with, a task allocation unit. For the purposes of this analysis we will group the two into a unified local-DMA and task-distribution unit. Each accelerator's local memory is typically connected to main memory through another (global) DMA engine. Depending on the context in which the accelerator was developed, it may also contain other specialized peripheral circuitry, such dedicated scaling or ReLU units.

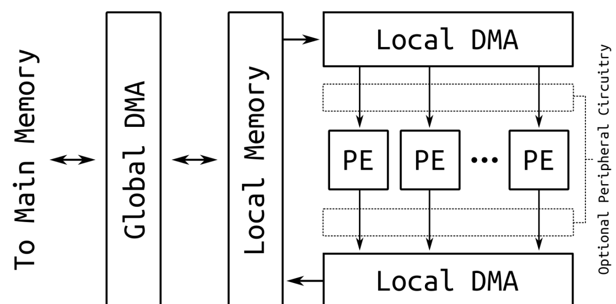

FIG. 2. The high level organization of sparse accelerators.

In this section, we will examine three accelerators targeting some form of sparse matrix multiply, two of which specifically address the power-law distribution of adjacency matrices.

A. AWB-GCN

The first accelerator of interest, AWB-GCN, specifically targets GCNs and focuses entirely on SpMM, arguing that the 70% density typically observed in output features is sparse enough to ignore GeMM [2]. The core of the AWB-GCN accelerator consists of a 1D array of PEs fed by a Task Distributor Queue (TDQ). The TDQ is fed from two memories: a Sparse Matrix Memory (SpMMeM) containing ultrasparse matrices in the CSC format and a Dense Column Memory (DCM) containing dense and moderately sparse matrices. The decision to store moderately sparse memories in dense format was motivated by the fact that below a certain sparsity there is more overhead storing a matrix in a sparse format than there is to keep them in dense format. The PE array feeds into a third memory, the accumulator, which collects and accumulates outputs from the PEs until they are ready to be written back to another memory.

The dataflow in this accelerator is based on a column-wise product, as one column of output is generated at a time:

$$C = A \cdot B \tag{6}$$

$$C[:, i] = \sum_k B[k, i] \cdot A[:, i], \tag{7}$$

where A is sparse, B is dense, and C is the output. The TDQ distributes the workload over each column of A. That is PE_0 may compute $C[0:2,0] = \sum_{k=0}^{2} B[k,0] \cdot A[0:2,0]$, while PE_1 computes $C[3:4,0] = \sum_{k=3}^{4} B[k,0] \cdot A[3:4,0]$ and so on. In addition to the basic task of workload scheduling, the TDQ monitors each PE and rebalances the workload for optimal performance. Depending on whether A is sparse or ultra-sparse, one of two different TDQs will be used, the first pulling A from DCM and the second pulling A from SpMMeM. The overall function of both TDQs remains the same, though the implementations differ due to the different input types.

To manage address workload imbalances due to power-law distributions—which causes some columns to be much denser than others, the accelerator includes an arbiter unit that manages workload distribution and balancing over groups of PEs. To manage what the authors call "evil rows," rows that are too dense to be balanced by the other techniques, the accelerator also includes row-remapping support. All of this—the TDQs, the arbiter, the autotuner that detects "evil rows—" everything between the memories and PE array, comprises what we have termed the local DMA.

B. MatRaptor

The second accelerator in our analysis, MatRaptor focuses on a sparse-sparse matrix multiply (SpGeMM), taking a row-wise approach similar to AWB-GCN's column-wise product [10]:

$$C = A \cdot B \tag{8}$$

$$C[i, :] = \sum_k A[i, k] \cdot B[k, :], \tag{9}$$

where both A and B are sparse, and C is the output. In MatRaptor, each PE operates on a full row of A producing a full row of the output C. Each PE contains three 4KB queues that are used to store partial products for a row, and each PE manages its own multiply-merge rhythm.

To prevent having to halt PEs when rows inevitably do not have the same sparsity, MatRaptor elects to have the PEs operate asynchronously, each receiving a new workload once the old one is complete in a round-robin fashion. With asynchronous PE operation, comes the potential for memory channel conflicts when two PEs try to access data in the same bank. This is resolved by the development of a new sparse format: C^2SR. This format is something akin to a hardware-aware CSR format, wherein each row is mapped to a unique channel. The result is that no two PEs ever need to access the same memory channel for matrices A or C. Since the allocation of rows to channels is a direct function of their sparsity, this arrangement also solves the power-law distribution problem.

The increased complexity of PEs in MatRaptor and tailored sparse format results in a more straightforward DMA that is spared the task of assigning workloads—since rows are inherently assigned to PEs in the C^2SR format. The DMA in MatRaptor consists of a crossbar connecting the PE array and two arrays of Sparse Matrix Loaders to a high-bandwidth memory (HBM). Each PE has its own dedicated loaders for the A and B matrices and writes its own outputs directly to the HBM.

C. SIGMA

The final accelerator in this overview is SIGMA, a generalized matrix-multiply accelerator focused on DNNs. The design of SIGMA is focused on being able to run GeMM and SpGeMM efficiently on regular and irregularly shaped matrices. Like the other accelerators, SIGMA consists of a 1D array of PEs, and like MatRaptor they are more complex than PEs in a traditional systolic array. Each PE consists of a 1D array of multipliers connected to an adder reduction tree. The inputs to the PE are routed through a benes distribution network to the multipliers to construct a flexible fabric. This flexible fabric allows the accelerator to support a near infinite number of dataflows, at the cost of a much more complex local DMA. With its focus on DNNs however, the design of the accelerator ignores ultra-sparse matrices, and elects to use a bitmap storage format for its sparse matrices.

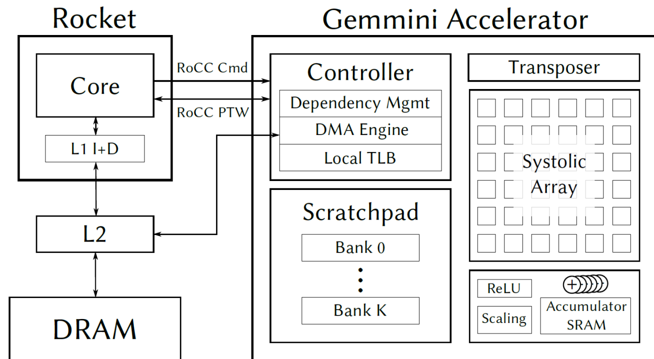

FIG. 3. Architecture of the Gemmini Accelerator, figure from Genc et. al. [1]

D. Relation to our Work

Despite the variety in the target kernels and the context of these accelerators, they all share a similar high-level organization. Each accelerator consists of a 1D array of PEs of varying complexity. This processing array is fed by a local DMA that interfaces with local memory and manages workload distribution. The outputs of the processing array feed directly into a local memory. This high-level organization is similar to Gemmini, a configurable generator for systolic-array-based matrix multiplication accelerators written in Chisel [1].

The Gemmini architecture, shown in fig. 3, includes a DMA Engine that connects the L2 cache of the host system to the accelerator's scratchpad. This feeds input matrices to the systolic array, which is currently capable of output-stationary and weight-stationary dataflows. Outputs from the array can be accumulated in the Accumulator SRAM, which functionally serves as an extension of the scratchpad memory. The generator can be configured to include additional units including a transposer, a ReLU unit, and a scaling unit for quantized models, if desired. The main difference between Gemmini and the high-level organization of other accelerators is the replacement of the systolic with a 1D array of PEs. While the systolic array in Gemmini is fed from two sides, with inputs propagating from one PE to another, the 1D array of PEs in sparse accelerators is fed by and feeds into the the local DMA directly. This difference can be bridged by adding connections between each PE in the systolic array directly to the local DMA, essentially providing an option to flatten the array, and creating a DMA that can feed the resulting array. In this context, it is not difficult to envision adding configuration options to Gemmini that would enable SpMM support.

IV. CHARACTERIZING THE COMPUTATION

Moving forward, we will consider GCNs consisting of two renormalized graph convolution layers, with ReLU as the nonlinear activation:

$$Out = \sigma\left(\hat{A} \cdot \sigma\left(\hat{A} \cdot FW_0\right) W_1\right), \tag{10}$$

where σ is the ReLU function, F_{in} is the number of input features, H is the number of features in the hidden layer, F_{out} is the number of output features, $W_1 \in \mathbb{R}^{F_{in} \times H}$ are the weights of the first layer, and $W_2 \in \mathbb{R}^{H \times F_{out}}$ are the weights of the second layer. Unless explicitly stated, we will use a subset of the citeseer dataset containing the first 1000 articles, with $F_{in} = 3703$, $H = 300$ and $F_{out} = 6$.

Since Gemmini supports running ONNX models [6] and Pytorch supports exporting ONNX models, we chose to use the sparse format that Pytorch uses—the COO format—for all of our sparse matrices. The COO format represents sparse matrices with two tensors: a 1D tensor containing all nonzero values, and a 2D tensor giving the coordinates of each corresponding nonzero value.

$$Indices = \begin{bmatrix} r_1 & r_2 & r_3 \ldots & r_n \\ c_1 & c_2 & c_3 \ldots & c_n \end{bmatrix} \tag{11}$$

$$Values = \begin{bmatrix} v_1 & v_2 & v_3 \ldots & v_n \end{bmatrix} \tag{12}$$

It is worth noting that the differences between COO, CSR, CSC, and similar sparse formats are not significant enough to fundamentally change these results.

In a GCN there are four primary operations: sDMA, dDMA, SpMM, and GeMM. The sparse DMA is used to load the renormalized adjacency matrix and input features, while the dense DMA is used to load the weight matrices and the outputs of the previous layer. It should be noted that even if the previous layer's outputs are sparse, using a dDMA may still be more efficient than a sDMA, due to the overhead of storing index arrays. The exact crossover point depends on the types used for storing indices and values, but is typically around 50% to 66% density. For this analysis, however, we will assume that the outputs of each layer are indeed dense. As for SpMM, this is used for both matrix multiplications in the first layer, and the multiplication of the adjacency in all subsequent layers. Meanwhile the product of features and weights $F^{(i)}W^{(i)}$ will use GeMM in all but the first layer.

Figure 4 shows the total time spent per operation on our example 2-layer GCN. For this GCN the number of hidden features is 300, data is stored as 32-bit floating point, and indices are stored as 32-bit integers. We assume 100% utilization of a 16×16 systolic array. The slow DMA bars reflect a typical worst-case memory throughput of 8 bytes per cycle, while the fast DMA reflects the typical best case of 16 bytes per cycle. At this data point, we can see that the runtime of the GCN is dominated by sparse multiplications when SpMM is not supported, followed by sparse dma when sDMA is unsupported. We

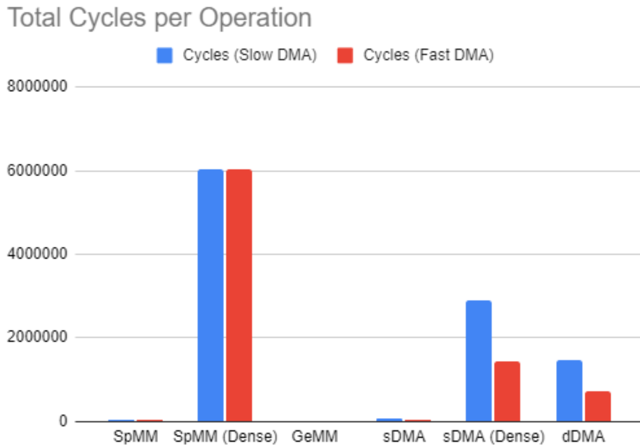

FIG. 4. Cycles spent per operation on a 2-layer GCN using a 1000 node subset of the citeseer dataset.

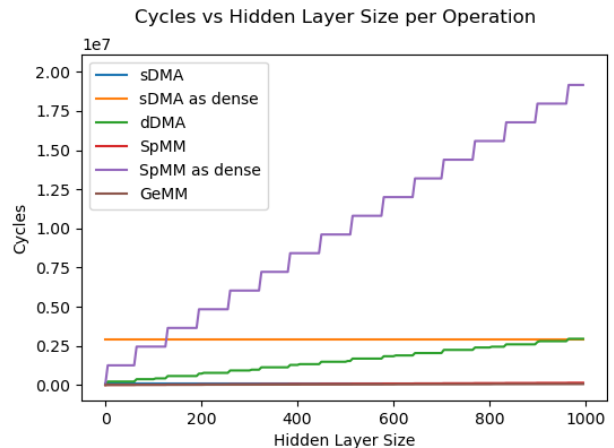

FIG. 5. **Cycles per operation as a function of hidden-layer size.** Note that sDMA dominates for small hidden-layers and SpMM dominates in large ones.

can also see that dense multiplications are relatively insignificant as parts of this computation. This is mainly due to the fact that of the four matrix multiplications in this 2-layer GCN only one is a GeMM, and a relatively small one at that. In GCNs with more layers, dense matrix multiplications are expected to weight more heavily, though they will remain not as important as SpMM and sDMA.

To get a better understanding of how the properties of the GCNs and systolic arrays affect the relative importance of implementing sDMA and SpMM, we plot a few key parameters against the cycles consumed by each operation. In particular we are interested in the cycles consumed by sDMA and SpMM when these operations are not implemented—that is when these operations must run on their dense counterparts—in comparison to the cycles consumed by dDMA and GeMM. To distinguish between the cycles consumed by sDMA and SpDMA when run on their dense counterparts and when properly implemented, we refer to the former "as dense". Except for the analyzed parameter, the following tests use the same GCN and systolic array configuration used above. These tests assume perfect overlap of DMA and matrix-multiplication, a scratchpad size of 128KB, an accumulator size of 32KB, an L2 cache size of 512KB, and a DMA speed of 8 bytes per cycle.

In Figure 5, we consider the effect of hidden layer size on these four operations. When neither SpMM nor a sparse DMA are implemented, we see that for small hidden layers the runtime is dominated by the dense loading of sparse matrices, in particular the large adjacency matrix. The size of these matrices, however, is unaffected by the size of the hidden layer, so when the hidden layer grows in size, so does the runtime of the GeMM between the output of the first layer and the weights of the second layer. The final sparse multiplication also grows proportionally to the size of the hidden layer. As a result, we end up with a crossover point where runtime switches from being limited by the lack of sDMA, to being limited by the lack of SpMM. If we increase the size of the input graph, we shift this point to the right, as our adjacency matrix grows proportionally.

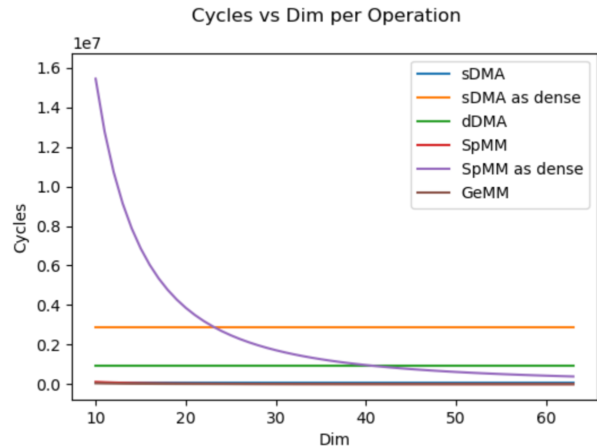

FIG. 6. **Cycles per operation as a function of systolic array size.** Note that SpMM dominates in small arrays and sDMA dominates in large arrays.

In Figure 6 we observe a similar pattern when considering the dimensions of the systolic array itself. While the speed at which we can perform matrix multiplication grows by the square of the dimension of the systolic array, the amount of data that needs to be moved in and out remains unchanged. As a result, large systolic arrays suffer more heavily from under-utilization due to a slow DMA, while smaller arrays find themselves compute-limited.

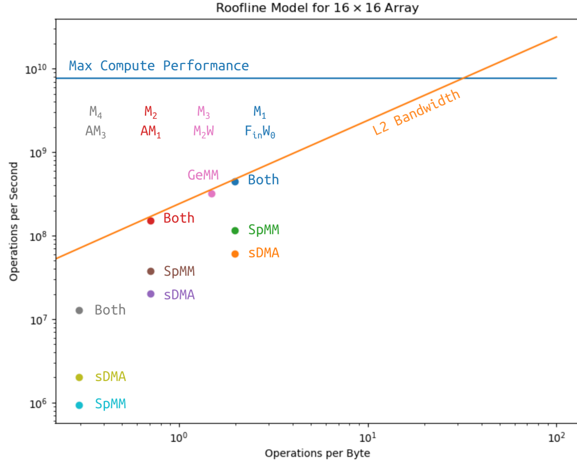

FIG. 7. **Roofline Model** for a 16 × 16 systolic array.

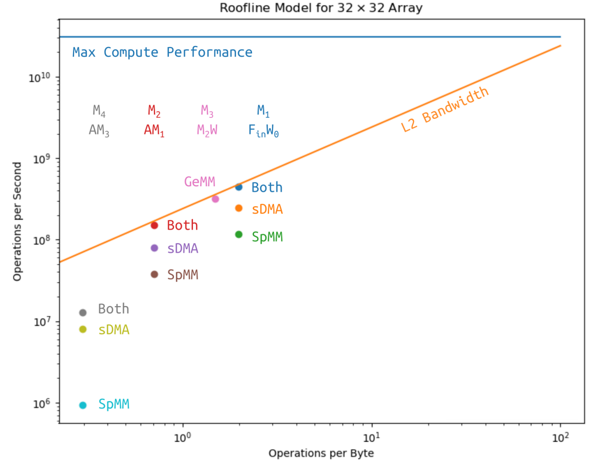

FIG. 8. **Roofline Model** for a 32 × 32 systolic array.

Putting the entire system together, we observe predicted roofline models in figs. 7 and 8 for a 16 × 16 and 32 × 32 systolic array respectively. Here, we plot performance of each matrix multiplication in eq. (10) in three cases: with sDMA available, with SpMM available, and with both available. When both optimizations are implemented, we find that we hit the roofline limits in all multiplications except M_4 due to its extremely narrow second argument. We are generally memory bandwidth limited in these cases, since a 16 × 16 floating point systolic array can consume 128 bytes of data per cycle and Gemmini's only receives 16 bytes per cycle in the best-case. These memory limits persist even when using 8-bit integers, which reduces memory consumption of the array to 32 bytes per cycle. When either sparse operation is not implemented, however, we fail to reach the roofline presented by the system architecture, and instead find ourselves limited by inefficient DMA or matrix multiplication. As before, we see that smaller systolic arrays are typically limited by inefficient compute, while larger arrays are limited by inefficient DMA, and it takes both optimizations to approach the roofline limits.

V. SPIKE DMA IMPLEMENTATION

In this work we focus on laying the groundwork for a sparse DMA that decompresses sparse formats (specifically the COO format) writing them to the Gemmini scratchpad in dense format. The typical flow for adding new functionality into Gemmini begins with defining the software interface, creating software model of the implementation in Spike—the RISC-V ISA Simulator—, and verifying that implementation with baremetal C tests. The Spike implementation and baremetal tests will then serve as the golden reference when developing and verifying RTL.

A. Software Interface

The software interface for the sparse DMA is implemented as a pair of custom RISC-V instructions. The first instruction configures Gemmini with the addresses of data and index arrays. The second instruction starts the DMA and provides Gemmini with the start row and column indices, the number of rows and columns to load, and the scratchpad address to write the dense expansion. These two instructions are bundled into a macro `gemmini_extended_mvin_sparse_coo(dataAddr, indexAddr, scratchpadAddr, startCol, cols, startRow, rows)` to simplify software implementation.

B. DMA Algorithm

The expanding DMA algorithm is fairly straightforward. We iterate through the row and column ranges of interest checking if each coordinate can be found in the index array. If the coordinate is found, we write the corresponding data value to the correct location in scratchpad or accumulator SRAMs. If the coordinate is not found, a zero is written. A simplified version of the algorithm is included below.

function SPARSECONFIGURE(dataAddr, indexAddr)
  $gemminiState.dataAddr \leftarrow dataAddr$
  $gemminiState.indexAddr \leftarrow indexAddr$
end function

function SPARSEMVINCOO(spAddr, data)

```
    dataAddr ← gemminiState.dataAddr
    indexAddr ← gemminiState.indexAddr
    toAccumulator ← spAddr[31]
    accumulate ← spAddr[30]
    baseSpAddr ← spAddr[28 : 0]

    cols ← spAddr[47 : 32]
    rows ← spAddr[63 : 48]
    colStart ← data[15 : 0]
    rowStart ← data[31 : 16]
    nextRow ← ReadDram(indexAddr)
    indexAddr ← indexAddr + indBytes
    nextCol ← ReadDram(indexAddr)

    for row ∈ {rowStart → rowStart + rows} do
        for col ∈ {colStart → colStart + cols} do
            block ← col / DIM
            spCol ← col % DIM
            value ← 0
            if row = nextRow&col = nextCol then
                value ← ReadDram(dataAddr)
                dataAddr ← dataAddr + dataBytes
                indexAddr ← indexAddr + indBytes
                nextRow ← ReadDram(indexAddr)
                indexAddr ← indexAddr + indBytes
                nextCol ← ReadDram(indexAddr)
            end if
            spAddr ← baseSpAddr + row + block ∗ DIM
            if toAccumulator then
                WriteAcc(spAddr, spCol,
                         accumulate, value)
            else
                WriteSp(spAddr, spCol, value)
            end if
        end for
    end for
end function
```

VI. BASELINE RESULTS

As a proof of concept, and to get baseline numbers, we run the 2-layer GCN described in section IV with $H = 300$ on the first 1000 nodes of the citeseer dataset. This GCN uses dense operations for all matrices involved and was written in PyTorch and exported to ONNX. The runner for the ONNX models was written in C based on the work of Prakash [6], and compiled for FireSim. The ONNX model was then run on a 16×16 Gemmini FP32 configuration at 30MHz. The results of these baseline tests are summarized in Table II.

Testing in FireSim—an FPGA accelerated cycle-accurate simulation platform—revealed an interesting issue: it appears that the ONNX model incurs significant overhead beyond the necessary matrix multiplications. As a systolic array, Gemmini should easily have at least 30% utilization on GeMM, but to end up with a mere 0.03% indicates that something else is going on. The root cause of this issue is currently unknown. The ONNX model is known to be functionally correct, and inference on the GCN through ONNX has been consistently successful. Running the same binaries in Spike yields similar results, indicating that the overhead is not part of the matrix-multiplication, as Spike has tendencies to under-estimate the cycle counts of multi-cycle CISC-like instructions. To circumvent this issue with the ONNX models and obtain reasonable baseline numbers, we decided to model the GCN as through its component matrix multiplications in C.

MAC Cycles (Dense)	5541797
MAC Cycles (Sparse)	47817
ONNX Model Cycles	16158006157
ONNX Dense Utilization	0.03%
ONNX Sparse Utilization	0%

TABLE I. Baseline FireSim Results: ONNX Model

MAC Cycles (Dense)	5541797
MAC Cycles (Sparse)	47817
Predicted Cycles	6569430
Predicted Dense Utilization	84.35%
Baremetal Cycles	10942891
Baremetal Dense Utilization	50.64%
Baremetal Sparse Utilization	0.44%

TABLE II. Baseline FireSim Results: Series of Matmuls

The results from the C model, shown in table II, reach 50% utilization when treating all multiplications as dense. This is not too far off from the predicted utilization of 84% when considering that the prediction assumed perfect overlap, and may have used different tiling dimensions. When we do account for the sparsity of the adjacency matrix and input features, however, this utilization drops significantly to about 0.4%. With the implementation of the sDMA proposed in Section V, we predict that this utilization will jump 0.8%. This may seem insignificant, but if we look at this GCN in the context of Figure 5, we notice that this GCN, having 300 features in its hidden layer, is dominated by SpMM rather than sDMA. This same GCN, when run on a theoretical Gemmini with SpMM support, would have a predicted utilization of 1.1%. With both sDMA and SpMM support, the theoretical utilization jumps to 31%, revealing the importance of both operations working in tandem.

VII. FUTURE WORK

The immediate next steps for this research include the implementing the proposed sparse-to-dense DMA in RTL and integrating it into the Gemmini architecture. An efficient implementation of the DMA will need to send requests for sparse index-data pairs in batches. A buffer will be necessary to buffer memory responses, since re-

quests may return out of order. The simplest DMA will only write to the scratchpad in order, but a trade can be made for efficiency as writes to the scratchpad can be done as soon as two consecutive index-value pairs are known.

After the implementation and verification of the expanding DMA, the next steps would be adding SpMM support to Gemmini. This will require making choices about how complex the sparse-capable PEs should be, or if the complexity will be shouldered by the local DMA. More complex PEs may reduce dense performance when Gemmini is configured to support both, while a more complex DMA will be more prone to RTL errors and may be more difficult to manage, since the local DMA also plays the role of task allocation. Depending on the complexity of the PEs and the chosen dataflow, the connection between the PE array and the accumulator may need to become more complex, possibly resembling a crossbar.

The architectural choices involved in adding SpMM support will seriously influence each other, and can be heavily informed by the choice of sparse format. In this regard, the C^2SR format paired with a row-wise product approach is quite promising in its ability to dramatically simplify the complexity of the DMA engine and task allocation. Since our interest concerning GCNs is focused on SpMM and not SpGeMM, the complexity of PEs can be much simpler than the MatRaptor PE design, as the density of the second argument removes the need for merging partial sums. With this approach, changes to PE and accumulator design would be relatively straightforward, and the main challenge would probably be revolve around Gemmini's ability switch between SpMM and GeMM at runtime, as SpMM prefers seeing a 1D array of PEs, while GeMM prefers a 2D array.

REFERENCES

[1] Hasan Genc et al. "Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures". In: *arXiv preprint arXiv:1911.09925* (2019).

[2] Tong Geng et al. "AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing". In: *arXiv e-prints*, arXiv:1908.10834 (Aug. 2019), arXiv:1908.10834. arXiv: `1908.10834 [cs.DC]`.

[3] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *International Conference on Learning Representations (ICLR)*. 2017.

[4] Omer Nagar et al. "Quadratic GCN for Graph Classification". In: (Apr. 2021). eprint: `https://arxiv.org/pdf/2104.06750v1.pdf`.

[5] Subhankar Pal et al. "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 724–736. DOI: `10.1109/HPCA.2018.00067`.

[6] Pranav Prakash. "End-to-end Model Inference and Training on Gemmini". In: (May 2021). eprint: `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-37.pdf`.

[7] Eric Qin et al. "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 58–70. DOI: `10.1109/HPCA47549.2020.00015`.

[8] Luana Ruiz et al. "Invariance-Preserving Localized Activation Functions for Graph Neural Networks". In: *IEEE Transactions on Signal Processing* 68 (2020), pp. 127–141. ISSN: 1941-0476. DOI: `10.1109/tsp.2019.2955832`. URL: `http://dx.doi.org/10.1109/TSP.2019.2955832`.

[9] Santiago Segarra et al. "Attributing the Authorship of the Henry VI Plays by Word Adjacency". In: *Shakespeare Quarterly* 67.2 (Apr. 2016), pp. 232–256. ISSN: 0037-3222. DOI: `10.1353/shq.2016.0024`. eprint: `https://academic.oup.com/sq/article-pdf/67/2/232/26707288/sq0232.pdf`. URL: `https://doi.org/10.1353/shq.2016.0024`.

[10] Nitish Srivastava et al. "MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 766–780. DOI: `10.1109/MICRO50266.2020.00068`.