8×8 Weight Block Sparsity: Training, Compilers, and FPGA-AIE Accelerators

Anonymous

***Abstract*—We present the main ideas about a vertical system where convolution and matrix multiplication weights can be trained to exploit an 8x8 block sparsity, compilers recognize it for both data compaction and computation reduction by a coherent splitting into threads. If we take a Resnet50, we can reduce the weight by half with little accuracy loss. We can achieve speeds similar to an hypothetical Resnet25. We shall present performance estimates by accurate and complete code generation for AIE2 configuration sets (AMD Versal FPGAs) using Resnet50, Inception V3, and VGG16, in order to highlight necessary symbiosis between HW overlay designs and the software stack to compile and to execute machine learning applications.**

I. Introduction

Block sparsity is an intuitive concept but it can be misunderstood. Take a matrix multiplication in Equation 1

$$\begin{pmatrix} \boldsymbol{C}_0 & \boldsymbol{C}_1 \\ \boldsymbol{C}_2 & \boldsymbol{C}_3 \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}_0 & \boldsymbol{A}_1 \\ \boldsymbol{A}_2 & \boldsymbol{A}_3 \end{pmatrix} \begin{pmatrix} \boldsymbol{0} & \boldsymbol{B}_1 \\ \boldsymbol{B}_2 & \boldsymbol{0} \end{pmatrix} \tag{1}$$

This is the computation

$$\boldsymbol{C}_0 = \boldsymbol{A}_1\boldsymbol{B}_2;\ \boldsymbol{C}_1 = \boldsymbol{A}_0\boldsymbol{B}_1;\ \boldsymbol{C}_2 = \boldsymbol{A}_3\boldsymbol{B}_2;\ \boldsymbol{C}_3 = \boldsymbol{A}_2\boldsymbol{B}_1 \tag{2}$$

and in general with proper γ_i (i.e., a mask)

$$\boldsymbol{C}_i = \sum_{k=0}^{1} \boldsymbol{A}_{i+k}(\gamma_{2*k+i}\boldsymbol{B}_{2*k+i}) \tag{3}$$

Where the matrix $\boldsymbol{B}$ is constant, diagonal, and each sub-matrix $\boldsymbol{B}_2$ and $\boldsymbol{B}_1$ can split further down and may have even smaller zero blocks. In this work, we chose the basic block of $\boldsymbol{B}_i = 8 \times 8$. It is the smallest for architectures based on AMD AIE2 products and we support others (i.e., they are parameters). This is a well known data structure in the sparse computation field. We can use *compress block row* (CBR) or column format (CBC) or generalization of the coordinate format (COO). There are standard matrix sparse-matrix multiplication interfaces and algorithms for CPU and GPUs using this data format (where only one operand is sparse or both) [1], [2]. There is no counterparts for AIE2 as today but they are in the works.

Block sparsity is not found naturally in CNN models. We explore training techniques (PyTorch and the Keras). The most successful is the simplest. We take a pre-trained model. We compute a mask Γ of zeros/ones per layer by zeroing the more likely blocks (using a Norm). Then we train the model till convergence or accuracy are achieved. We take the sparse model and we quantize to 8-bit integer computations with the Vitis-AI quantizer. The final model is a XIR quantized model (Xilin intermediate representation). See Section III. We have a custom compiler that takes the XIR model and an abstraction of a connected set of AIE2. See Section IV. The compiler computes the maximum sub-volume computation per core. By heuristics and following a schedule, it computes a memory allocation in memtile (i.e., intermediate scratch pad) for input, outputs, and weights . It formats, compresses, and organizes the weights exploiting spatial distribution to memtiles and cores. We generate all the explicit communications between DDR (L_3), memtile (L_2), and cores (L_1). These are Gather and Scatter instructions with a complete and parametric estimate of their execution time by bandwtidh constraints and number of channels. We know the subproblem sizes per core, the computation throughput and with a clear specification of what is executed in parallel. Then, we can estimate the execution time per layer and entire network with an accuracy closer to a simulation. We will show estimates for three CNN models and eight different AIE designs; see Section VI. To our knowledge, we are the first in applying sparsity on AIEs systematically. The ability to provide different algorithms and easy to obtain estimates for very different configurations, it will allow to explore optimizations like sub-graph depth-wise tiling we could not have otherwise.

In our context, convolution is our main computation and CNN are networks we can train reasonably. This is because of legacy, we want to speed up the FPGA work-horses, convolutions provide more difficulties than GEMMs (padding, strides, and overlap), have usually biases with different precision requirements (weight 8bits and bias 32), routinely they can deploy different scaling factors per output channels, and GEMM is transformed into a 1×1 convolution immediately (the other way around is possible with proper activation preparation a convolution can be represented as a GEMM but the activation tensors explode and the preparation is more suitable for CPUs than FPGAs, but we have digressed).

In the following Section II, we start with a quantitative measure about the advantages of block sparsity.

II. Block-Sparse Size Matters

Consider $\Gamma, \Omega \in \{0,1\}^{N\times N}$ and $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{N\times N}$:

$$\boldsymbol{C} = (\Gamma\boldsymbol{A}) * (\Omega\boldsymbol{B})^t \tag{4}$$

More precisely, consider non-zero blocks of size $k \times k$ so that

$$\boldsymbol{C}_{i*N+j} = \sum_k (\gamma_{i*N+k}\boldsymbol{A}_{i*N+k})(\dot{\omega}_{j*N+k}\dot{\boldsymbol{B}}_{j*N+k}) \tag{5}$$

Thanks to the sparsity and if we store only non-zeros, then γ_{i*N+k} and $\dot{\omega}_{j*N+k}$ are contiguous (so their counterparts of $\boldsymbol{A}$ and $\boldsymbol{B}$). There will be a meaningful product to compute if and only if $\gamma_{i*N+k} = 1$ and $\dot{\omega}_{j*N+k} = 1$. We merge-sort these

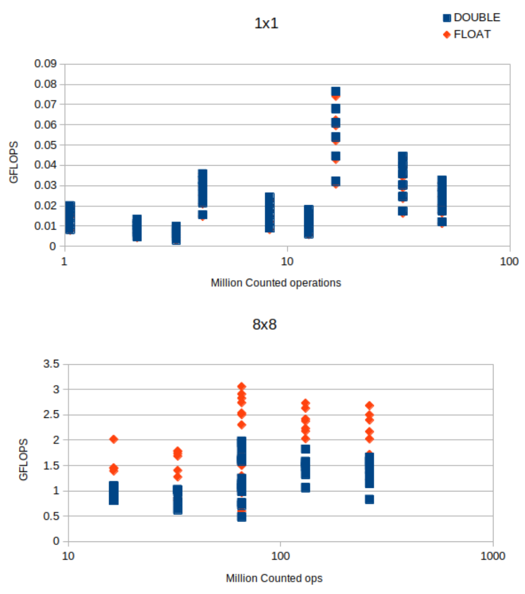

Fig. 1. Block 1x1 and 8x8 performance

vectors. See how the sparse-sparse matrix multiplication using *Coordinate list* (COO) is applied in Figure 1 where the entry in the list is a submatrix. We provide software to reproduce this [].

If we want to achieve a fixed sparsity of 50% for a square matrix of size N and if we can choose the block size $k \times k$. The larger k is, the smaller the overhead will be.

In Figure 1, we present two scatter plots: on the abscissa the effective multiplication-and-addition number, on the ordinate the performance in GFLOPS, when the sparse matrix with dense block is 1×1 (above) and 8×8 (below). Given the same problem, we deploy more threads (more GFLOPS). With the same number of effective operations, the block permits and exploits higher GFLOPS per effective operation (Float is 2x faster than Double precision and this can be emphasized further [3], [4] and [5]). We are working with operands of size 8 bits and the size 8×8 will exploit the maximum throughput (8 bits 256 multiply-add per cycle per AIE core, 16 bits 128 MAC, 32bits 64 MAC).

III. Block Sparsity: Training and Quantization

The block sparsity is not recurring naturally. We must train the network for it. A convolution has a weight tensor in four dimension: $\boldsymbol{W} \in \mathbb{R}^{c_{out} \times h \times k \times c_{in}}$. In the hyperplane of the h and k, we can simplify the weight as $\dot{\boldsymbol{W}} \in \mathbb{R}^{c_{out} \times c_{in}}$ and block sparsity can be simply described by a mask $\Gamma\dot{\boldsymbol{W}}$. Although, we speak of a 8×8 of non-zeros, this is in practice a $8\times h\times k\times 8$ block. For the matrix multiply $h = k = 1$. We explain the training process.

A. Keras

We shall provide a repository using Keras [6] where we implements the contents of this section [].

We target convolutions only and without quantization. The idea is simple: we take any model and we create a copy where we enhance the convolution with a (non-trainable) Γ. A convolution will have three parameters (saving the model into a different format). The forward computation is modified so that the weights used for convolution are $\Gamma\boldsymbol{W}$. We assume the backward computation (i.e., gradient) is done automatically from the forward definition. There is no need to change the bias. For example, we take Resnet50 from the Keras application repository, we start with a $\Gamma = 1$, and we trained one epoch using imagenet repository [7]. The goal is to choose Γ in such a way we achieve the required sparsity and the minimum loss in accuracy. We tested different approaches such as incremental, Fisher measure, Hessian, diagonal Hessian, and custom penalty losses. We will give full description where space is not a requirement.

B. Γ Chosen Once and Full Training Ahead: PyTorch

Take a convolution with $\Gamma = 1$ and weights $\boldsymbol{W}$. For each γ_i, this will be representative of a block $\boldsymbol{W}_i \in \mathbb{R}^{8\times h\times w\times 8} \sim \mathbb{R}^{8\times 8}$. We can choose the $\boldsymbol{W}_i$ using a measure of importance:

- $L_2 = \sqrt{\sum_k w_k^2}$ with $w_k \in \boldsymbol{W}_i$,
- $L_1 = \sum_k |w_k|$ as above,
- Variance $\sigma^2 = \frac{1}{64}\sum_k (w_k - \mu)^2$ with $\mu = \frac{1}{64}\sum w_k, w_k \in \boldsymbol{W}_i$ or $\frac{1}{N}\sum w_k, w_k \in \boldsymbol{W}$. In signal processing σ^2 is the power of the signal.

We can then sort them in ascending order. We set the first half to zero. Then we start re-training. We do this for the entire network or for one convolution at a time.

In Table I, we show the results by using one-time mask and full training: VGG-16, ResNet-50, Inceptionv3 on ImageNet20 (20 classes) and ImageNet1k (1000 classes). We use three samples per class for the validation accuracy for ImageNet1k data set; instead, we use 50 samples per class for ImageNet20. Fine-tuning sparse networks on the original ImageNet dataset [7] is expensive. To reduce the training time, we chose 20 classes (from the original 1000 classes) with the least number of images per class in the training data-set and this choice will affect the accuracy because there are fewer samples for re-training.

TABLE I
Accuracies of the Sparsity Models

Model	Dataset	Baseline	Sparsity		
		Acc.(%)	block	ratio (%)	Acc.(%)
Inception-v3	ImageNet1k	77.2	8x8	50	75.5
ResNet-50	ImageNet1k	76.7	8x8	50	74.6
VGG-16	ImageNet1k	70.6	8x8	50	69.7
ResNet-50	ImageNet20	96.1	8x8	25	95.1
ResNet-50	ImageNet20	96.1	8x8	50	92.0
ResNet-50	ImageNet20	96.1	8x8	75	87.1
ResNet-50	ImageNet20	96.1	1x1	25	96.0
ResNet-50	ImageNet20	96.1	1x1	50	95.6
ResNet-50	ImageNet20	96.1	1x1	75	93.5
VGG-16	ImageNet20	92.0	8x8	50	89.6
VGG-16	ImageNet20	92.0	1x1	50	92.3
VGG-16	ImageNet20	92.0	1x1	75	91.7

Classification accuracy on ImageNet1k drops by only 1 - 2% after applying 50% sparsity with a 8×8 block (this is without any quantization). We experiment with different block shapes such as 16×4 and 4×16 on ResNet-50, but the accuracy is slightly worse. Fine-grained sparsity (1×1 block or unstructured) does not sacrifice any accuracy (i.e., almost any). This is not equivalent to 2 over 4 (or 4 over 8) sparsity now available in GPUs, experiments show such constraints make the accuracy drop by more than 30 percentage points (we do not report these results).

IV. The Compiler and its Code generation for AIE

We take a PyTorch/Keras model, quantize it using Vitis AI, and create an intermediate representation that we call Xilinx Intermediate Representation (XIR). XIR is a graph where each node is an operation that reads tensors and writes one tensor. A convolution has one quantized input INT8 with a position where the fraction starts (power of two scale). It computes a tensor using the same layout and with a proper scale. The weights and bias are properties of the convolutions. They are tailored and laid out at compile time, they are $COUT \times h \times w \times CIN$ (like the caffe layout [8] []).

The main differences from our previous compilers are the parameterized representation of block sparsity, the capability to split tensors and computations accordingly to a parameterized representation of the architecture. Our HW abstraction is a Python class, describing a variety of systems. All weights are statically prepared into DDR and we move them explicitly towards the inner levels. Inputs and outputs have designated space in DDR. DDR can and it will be used for tensors spills. The memory allocation to memtile is basically coloring algorithms and some heuristics. In this architecture, we do not allow *streaming* of neither data nor weights (because they share space in memtile and input and output have different consumption/production rates).

A. AIE Hardware Abstraction

Although, we present a single HW, keep in mind this as a representative for presentation purpose.

See Figure 2, we work with a mesh of 4x4 AIE2 cores connected by 4 horizontal and 4 vertical interconnections. We present estimates for square 2x2, .. $i \times i$ .. 8x8 and rectangular shapes are in the works (4×1, 4×2, and 8×2 into a 8×8 with 2 memtiles per column). Each core has 8 banks memories for a total 64 KB. About six banks are used as input/output/weight buffers and two banks are used as temporary space for kernels. Each core can request and send data to its direct neighbors (if aware of connection and control but this utility is not used here). Double buffering using ping/pong is used for inputs and outputs.

There are four memtiles: each 512 KB and each is connected to one columns and its direct neighbor column, or it is connected to a row and its neighbor. The total amount of space is 2 MB. Memtile is a circular buffer to exploit more flexible allocation. Note a 2×2 architecture will have one memtile per column and a total of two memtiles (1 MB).

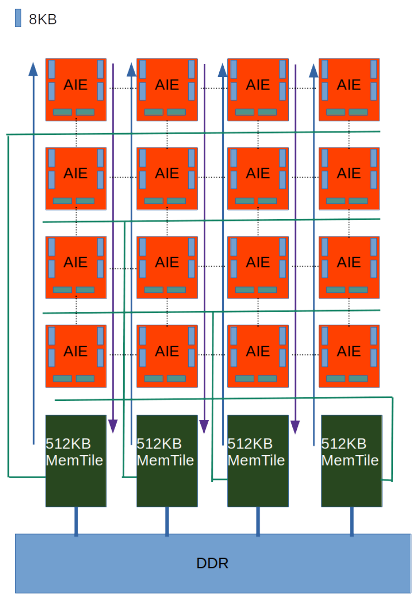

Fig. 2. 4x4 AIE representation

A Memtile can broadcast data per column or per row; it is a design choice. We can dedicate one memtile for weights, one for activations, or we can share it. In this work, we present results for shared memtiles. To maximize the computation parallelism, every core will write data per column into memtile.

B. Subvolumes, Data Compression, and Data Movements

The computation is split by memtile and thus by column (cores columns). The output tensor is computed and split evenly by width. Thus one memtile will store one partial tensor by width, each core will compute different output channels, and the computation streams the output tensor by rows and using ping/pong double buffering. We prioritize to reuse weights in core. The cores set is a cohort and we always choose symmetric computations. Activation sparsity of inputs and outputs are beyond the scope because we do not have any hardware support for the division of a sparse vector so that each core will have equivalent work. Also, we do not merge two operations like convolution and max-pool.

If we have the inputs, output, and weights in memtile, what is the largest computation we can do in the AIE? The minimum computation is one output channel and one row (i.e, by height). If this is not possible, we try to reduce the size of the width (e.g., shaping the tensor in memtile by using DDR spills) and we can manage to split the input channels and to split the weights accordingly and prepare for accumulattion. We call W-Split the distribution of tensor by columns in the AIE mesh. We can COUT-split, this requires the partial transfer of weights. We can CIN-split when we need to split by input channel, this is the last resort because it is also the most expensive (accumulation of the outputs). CIN split can be implemented as a graph optimization by splitting the convolution into two and then use an element wise operation to combine the results (this can be done recursively).

The subvolume describes the smallest shape of the weights that we need to manage and the largest computation in the core. We compress the weight accordingly. Any data movement will always be a multiple of the subvolume and is a single load. Such a compressed data will have the same properties whether it is sparse or dense. We address the optimal decomposition in Memtile and core-memory in a different venue.

C. Schedule and Memory Allocation

During the scheduling of each layer, we evaluate what tensors can fit in memtile. Here, activation and weight tensors share the space. At each step, the memory allocation will check if we can allocate (inputs, weights, and outputs). If we cannot, we evict all tensors into DDR and then split/time the computation.

At the end of this stage, every tensor will have an address in memtile or DDR (or both). If there are only DDR addresses, the compiler will take the basic layer computation and, by heuristics, will split the computation and the output tensor by width, output channel, height, and input channel (no necessarily in this order). The heuristics have a single objective to find the largest problem fitting the (each) memory level. We deploy a recursive approach of tiling. Formally, $\dot{\sum}$ is a parallel loop and a W-split can be written as follows:

$$\boldsymbol{Y} = Conv(\boldsymbol{X}, \boldsymbol{W}) = \dot{\sum}_{w} Conv(\boldsymbol{X}_w, \boldsymbol{W}) \tag{6}$$

The split is a function of the footprint. Before and after each convolution, there will be an explicit data movement (optional). At this stage each input, output, and weights have addresses associated with each sub-computation. Then the code generation of each $Conv(\boldsymbol{X}_w, \boldsymbol{W})$ is independent and recursive as needed. This is a tree. If the convolution has strides or a large kernel, each sub-convolution has overlap data; however, the sub-convolution has defined addresses and data movements. For a W-split such as this, we are computing the output by rows and the weights are reused (read once). Note scheduling and memory allocation we addressed them first.

D. Code Generation

The compiler creates a list of operations. These operations are smaller and smaller and they can be executed from memtile to memtile. There is a further decomposition using only AIE cores and it is completely determined by the subvolume. Here, we show how we generate code at this level and estimate time as in Figure 3. This is the computation of a convolution with top/bottom padding by height:

$$\boldsymbol{Y}_{height} = Conv(\boldsymbol{X}_{h=0}) + \dot{\sum}_{h=1}^{9} Conv(\boldsymbol{X}_h) + Conv(\boldsymbol{X}_{h=10}) \tag{7}$$

An important feature of the current system is the concept of **iteration** between memtile and core. Using locks and chaining the locks, we write a single instruction from the prospective of a single core (as a SIMD instruction) and driving all cores at once for multiple iterations $\dot{\sum}_{h=1}^{i} Conv(\boldsymbol{X}_w)$ in Equation 7, the ASM-like code follows:

```
LOADFM Lock k_0 memtile addr core addr iter i
CONV iteration      i
WRITEFM Lock k_1 memtile addr core addr iter i
```

There is an implicit lock (say `k_x`) that is used for the pong and the system cycles between locks (`k_x` and `k_0`). These three operations execute a number of iterations i and, using a ping/pong, they will load different slices of data and compute different slices of data.

Equation 7 is encoded as follows:

```
## Head top pad < 50 us First comp block
LOADFM Lock k_0 memtile addr_0 core addr iter 1
CONV iteration 1
WRITEFM Lock k_1 memtile addr_1 core addr iter 1
## Body iteration > 50 us < 150 us
## k_0 -> k_2 -> k_4 Lock Chain
LOADFM Lock k_2 memtile addr_2 core addr iter 9
CONV iteration 7
WRITEFM Lock k_3 memtile addr_3 core addr iter 9
## tail bottom pad > 150 us Last computation block
LOADFM Lock k_4 memtile addr_4 core addr iter 1
CONV iteration 1
WRITEFM Lock k_5 memtile addr_5 core addr iter 1
```

We present in Figure 3 the execution estimate of this code. At this stage, we have all the information. Per layer, the code generation is a two pass process. First, we generate code for the all loads/stores. Second we combine them into chains with dependency, logically correct and as fast as possible.

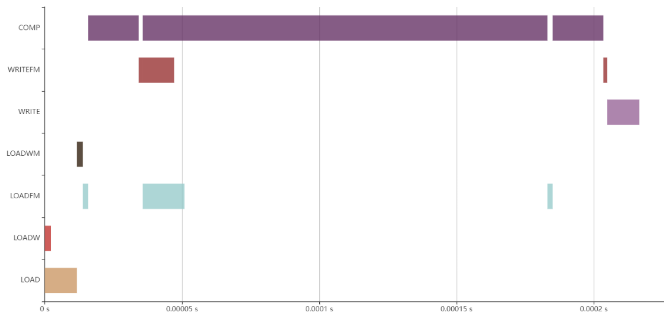

Fig. 3. Resnet single convolution with padding for 4x4: legend AIE: LOAD activation from DDR to memtile, LOADW weights from DDR to memtile, LOADFM activation from memtile to AIE2 cores, LOADWM weights from memtile to AIE2, WRITE from memtile to DDR, WRITEFM from AIE2 to memtile, COMP Computation.

We could estimate the time execution without a full code generation. When we annotate time information to a load, we have assurance that the load is a complete description of the DMA communication between multiple memories and as complex as the architecture. Actually, this is literally translated to a binary executable that perform the data movement.

E. Time Estimation

We explain how we capture the execution time and visualize it as in Figure 3. We start by the time estimates for DDR to memtile communications. We have two communication types: activations and weights. Per memtile there are two dedicated channels.

- If we share activations and weights in the same memtile, we can use one channel for activations and one for weights. Thus the loads from DDR to memtile (LOAD and LOADW) are parallel with a bandwidth of 4 GBps.

Writes from memtile to DDR (WRITE) can use both channels (8 GBps).
- If activations and weights go to different memtiles (for example weights to memtiles ’0’ and ’3’ and activations to ’1’ and ’2’), each load is parallel and 8 GBps. Writes are identical.

The memtile connections with AIE cores are different. We assume a few channels with again 4 GBps bandwidth. One memtile can broadcast inputs to a cores column. These communications are for activations (LOADFM). One memtile can broadcast to rows of cores, these are for weights (LOADWM). We assume that the column and row communications are parallel.

Every communication with iteration one is synchronous and sequential. The load, convolution, and store is executed one after the other and every core is independent. For synchronization and for bookkeeping, we assume that AIE2 weights communications (from memtiles) to cores are synchronous and halting (LOADWM).

Every communication with iteration larger than one, we assume that load (LOADFM), computation (COMP), and store (WRITEFM) are executed in parallel and the overall execution time is the maximum of the estimated time multiplied by the number of iterations.

We estimate the execution time of a subvolume (COMP) by the number of operations divided by the maximum number of operations per cycle which is in our scenario is $4 \times 8 \times 8 = 256$ operations per cycle and 1 GHz frequency. Sparsity reduces computation and communication time.

We do not account the wait and synchronization which are necessary to reprogram the fabric. These are very expensive running on for a few milliseconds.

F. Sparse Convolution example

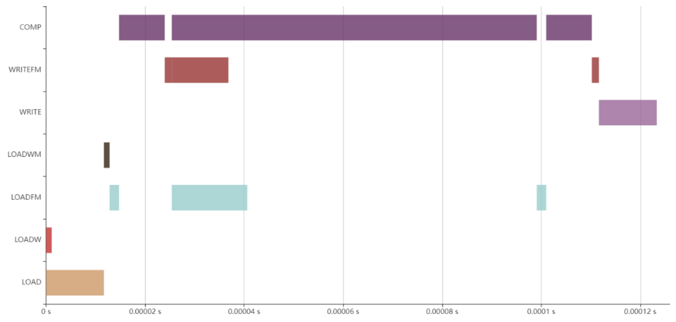

Fig. 4. Resnet single convolution with padding and sparsity for 4x4 AIE

We present the time estimate for a convolution with padding, dense, and with 50% sparsity, see Figure 3 and 4.

For these convolutions, the computation dominates the execution time. Sparsity cuts the execution time by half: from 200 μs to 130 μs. On one hand, there are convolutions that realize up to 2× performance; on the other, there are convolutions that are dominated by the reading or writing. In the latter case, sparsity helps in space saving and probably DDR tensors spilling. In principle, we could relax sparsity requirements for those convolutions that are communication bound (and restart training).

V. Depth-wise Tiling

Take the subgraph schedule $L_0, L_1, \dots L_j$, for every i L_i is a layer that produces a single tensor T_i. Let us define $u_j = 1 \times 1$, where we neglect the *channels*, as a sub-tensor of layer L_j; that is, a sub-vector of T_j. A projection $\P$ is a function taking u and projects the input sub-tensor needed to compute u. In reverse order, starting from L_j compute $\P(L_j, u_j)$ and propagate the projection. When a layer L_m and tensor T_m feeds two (or more) layers L_x and L_y. Then $u_m = \max(\P(L_x, u_x), \P(L_y, u_y))$ when u_x and u_y are defined completely and *max* mean the largest. We carry the propagation of $\P(L_m, u_m)$ and we have T_{-1}=$\P(L_0, u_0) = \P(L_0, \P(L_1, u_1)) = \P(L_0, \dots \P(L_l, u_l))$. At the end of the propagation, imagine this is a generalized convolution with kernel size $k = \P(L_0, u_0)$. Now if we repeat the process with $\dot{u}_j = 2 \times 2$ ... $k + s = \P(L_0, \dot{u}_0)$. Thus we have an estimate of the generalized stride. Now we can split the computation by selecting a non overlapping decomposition of the output tensor: say using M tiles each of size u_o. The input is split into M tiles of size $(u_o - 1)s + k$ and we will have an overlap of size $k - 1$. We can choose a tiling in such a way there is zero DDR communication in between L_0 and L_j. With input overlap, thus with extra reads, we have extra computations. Sparsity will improve weights communication and reduce the effect of computations. We use the term generalized convolution but it is not a convolution and it is applied to a graph computation (with element wise operations and transpose convolutions, we do not think it is applicable to height and width softmax or layer norms).

VI. Results

TABLE II
Execution Time Estimates

AIE2	Model	Dense sec	Sparse sec
2x2	Resnet	2.492347e-02	1.582626e-02
3x3		1.269543e-02	8.661490e-03
4x4		1.077318e-02	7.064918e-03
5x5		failed	4.303485e-03
6x6		5.712521e-03	4.490127e-03
7x7		4.205991e-03	3.212234e-03
8x8		6.376768e-03	4.602027e-03
2x2	IncV3	4.283837e-02	2.440544e-02
3x3		2.386600e-02	1.422390e-02
4x4		1.740967e-02	1.012540e-02
5x5		9.690552e-03	failed
6x6		1.063962e-02	6.439692e-03
7x7		8.727651e-03	failed
8x8		9.093276e-03	5.666152e-03
2x2	VGG16	4.476212e-02	2.608593e-02
3x3		2.53343e-02	1.002015e-02
4x4		1.371000e-02	8.852128e-03
5x5		failed	4.336479e-03
6x6		failed	5.770197e-03
7x7		7.455440e-03	5.288551e-03
8x8		9.203393e-03	6.502333e-03

In Table II, we present the performance of sparsity applied to all the convolutions (except the first one) for Resnet 50, Inception V3, and VGG16.

Corner cases are represented as failure in Table II. Some cases is because of inability to break the weight tensor evenly. Sometime is for incorrect data management especially for prime numbers. These are all issues will be address as the technology matures. Please, note that VGG16 using 8x8 configuration is slower than 7x7 (by using sparse). For a symmetric computation too small sub-volume computations make the computation overall more inefficient and requiring more iterations for the same amount of data transfers. This is a case where more HW does not improve performance, which is interesting and relevant.

A. Results Depth-Wise Tiling for VGG16 3x3

We take VGG and we instruct the DDR to be 16 times slower (4GBs/16) highlighting the need of fewer DDR communications. We take only the part that requires DDR spills for each layer: 0.025s total time. We apply depth-wise tiling using three tiles and we break even at 0.024s. With two tiles, we achieve better performance at 0.022s. Sparsity by itself without any tiling can achieve only 0.021s. Sparsity and tiling improves even further and we achieve 0.014s. A posteriori, we can appreciate the reduction of activation DDR communications thanks to depth-wise tiling and the reduction of computation and weights communication by sparsity.

We present results for VGG16 because of simplicity and each layer tensor do not fit the three memtile (of a 3×3 system). We can apply the same approach to Resnet and inception. The generalized convolution idea is applicable.

VII. Conclusions and Context

This is a multifaceted problem and we present a complete solution from training techniques, compilers, code generation, HW definition, and time estimations. It is a vertical software system, more complex than just a prototype, and it is used for the validation and comparison of different HW designs. A few convolutions have been validated in simulation and in hardware.

This could be seen as a quantization and sparsification problem. For example, how we can reduce the footprint of a CNN network. There are post training techniques that are targeting quantization and unstructured sparsity [9] and all the references within. We need to be more aggressive and training for it (as starting point [10]). Our sparsity is not really a property of the model, software can describe it, and the hardware can take advantage; however, we do not need specific hardware support at instruction level. To our knowledge we are the first applying sparsity to AIE2 overlays systematically.

This work stemmed from a collaboration between Xilinx and Numenta and the idea set forward by the authors in [11], where the models are too dense. The authors' contribution cannot be completely applied using AIE2 systems (no PL for costom operations).

The difficulty of generate code for complex architectures can be described and solved in different ways. There are recent attempts of introducing SW/HW solution for spatial processors like ours [12], [13], [14]. Usually major attention is given only to matrix multiplication and GPUs [3] [4], we can work on only static sparsity at this time. Matrix multiplication is appealing for the application in LLM and application in GPUs. Convolutions is far richer in complexity and it is the work-horse for FPGAs based products and systolic array systems/computation.

References

[1] AMD, "rocsparse," https://rocsparse.readthedocs.io/en/master/, 2020.

[2] NVIDIA, "cusparse," https://developer.nvidia.com/cusparse, 2020.

[3] S. Gray, A. Radford, and D. P. Kingma, "Gpu kernels for block-sparse weights," 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:52220661

[4] Z. Li, D. Orr, V. Ohan, G. D. costa, T. Murray, A. Sanders, D. Beker, and D. Masters, "Popsparse: Accelerated block sparse matrix multiplication on ipu," 2023.

[5] M. Kurtz, J. Kopinsky, R. Gelashvili, A. Matveev, J. Carr, M. Goin, W. Leiserson, S. Moore, N. Shavit, and D. Alistarh, "Inducing and exploiting activation sparsity for fast inference on deep neural networks," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 5533–5543. [Online]. Available: https://proceedings.mlr.press/v119/kurtz20a.html

[6] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[8] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 675–678. [Online]. Available: https://doi.org/10.1145/2647868.2654889

[9] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," 2023.

[10] B. Hawks, J. M. Duarte, N. J. Fraser, A. Pappalardo, N. Tran, and Y. Umuroglu, "Ps and qs: Quantization-aware pruning for efficient low latency neural network inference," *CoRR*, vol. abs/2102.11289, 2021. [Online]. Available: https://arxiv.org/abs/2102.11289

[11] S. Ahmad and L. Scheinkman, "How can we be so dense? the benefits of using highly sparse representations," 2019.

[12] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 554–566, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:233740109

[13] E. Russo, M. Palesi, G. Ascia, D. Patti, S. Monteleone, and V. Catania, "Memory-aware dnn algorithm-hardware mapping via integer linear programming," *Proceedings of the 20th ACM International Conference on Computing Frontiers*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:260442361

[14] J. Cai, Y. Wei, Z. Wu, S. Peng, and K. Ma, "Inter-layer scheduling space definition and exploration for tiled accelerators," *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259177591