Weight Block Sparsity: Training, Compilers, and Accelerators

P. D'Alberto, T. Jeong, A. Jain, S. Manjunath, M. Sarmah,
S. Hsu, Y. Raparti, and N. Pipralia

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 such a sparsity for both data compaction and computation 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 a small and efficient set of AIE2 (AMD Versal FPGAs) using Resnet50, Inception V3, and VGG16.

KEYWORDS

AI, FPGA, Performance, Sparsity, and Tools

ACM Reference Format:
P. D'Alberto, T. Jeong, A. Jain, S. Manjunath, M. Sarmah, and S. Hsu, Y. Raparti, and N. Pipralia. 2024. Weight Block Sparsity: Training, Compilers, and Accelerators. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym DAC).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

We explain what we mean for block sparsity and for a vertical solution. Block sparsity is an intuitive concept but it is also a little misunderstood. Take a matrix multiplication in Equation 1

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

This is the computation

$$\boldsymbol{C}_0 = \boldsymbol{A}_1\boldsymbol{B}_2;\ \boldsymbol{C}_1 = \boldsymbol{A}_0\boldsymbol{B}_1;\ \boldsymbol{C}_2 = \boldsymbol{A}_3\boldsymbol{B}_2;\ \boldsymbol{C}_3 = \boldsymbol{A}_2\boldsymbol{B}_1 \quad (2)$$

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

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

Where the matrix $\boldsymbol{B}$ is constant, diagonal, and each submatrix $\boldsymbol{B}_2$ 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 a great starting point for architectures based on AMD AIE2 products and we support others. For example,

$$\boldsymbol{B} = \dot{\sum_i} \gamma_i \boldsymbol{B}_i, \quad \gamma_i \in \{0, 1\} \quad (4)$$

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym DAC, June , 2024, San Francisco, CA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN TBD
https://doi.org/XXXXXXX.XXXXXXX

This is a well known data structure in the sparse computation field. We can use *compress block row* (CBR) or column format (CBC). 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) [2, 16].

We explore training techniques (PyTorch and the Keras). The most successful is the simplest. We take a pre-trained model. We compute a Γ per layer using a function to determine the more likely zeros blocks (using a Norm). Then we train the model till convergence or accuracy 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 3. We have a custom compiler that takes the XIR model and an abstraction of a connected set of AIE2. See Section 4. 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 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). 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 compute time estimates for all parts of the computation. We will show estimates for three CNN models and eight different AIE designs; see Section 6.

In the following Section 2, we start with a quantitative measure about the advantages of block sparsity.

2 BLOCK-SPARSE MATRIX-MATRIX MULTIPLICATION

As a thought experiment, consider Γ and Ω two matrices in $\{0, 1\}^{N \times N}$.

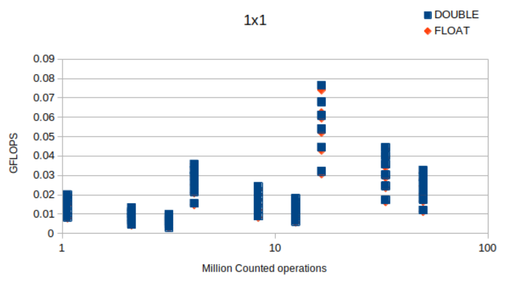
$$\boldsymbol{C} = (\Gamma \boldsymbol{A}) * (\Omega \boldsymbol{B})^t \quad (5)$$

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

$$\boldsymbol{C}_{i*N+j} = \sum_k (\gamma_{i*N+k}\boldsymbol{A}_{i*N+k})(\dot{\omega}_{j*N+k}\dot{\boldsymbol{B}}_{j*N+k}) \quad (6)$$

Thanks to the sparsity and if we store only non-zeros, then γ_{i*N+k} and $\dot{\omega}_{j*N+k}$ are contiguous. 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 vectors. See how the sparse sparse matrix multiplication using *Coordinate Block Structure* (COO) is applied in Figure 1. We provide software to reproduce this [5].

Now, assume we want to achieve a fixed sparsity (i.e., density) of 50% for a square matrix of size N and we choose the block size $k \times k$. The larger k is, the smaller the overhead will be. The relative performance of the k^3 multiplication is better as k get larger because spatial, temporal locality, and further optimized code for a constant/parameterized such as k.



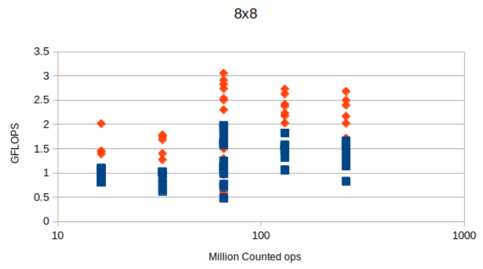


Figure 1: Block 1x1 and 8x8 performance

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 and thus the Jenga effect. 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 [10, 15] and [14]).

3 BLOCK SPARSITY: TRAINING AND QUANTIZATION

In Convolutional Neural Networks, the two main operations are convolutions/correlations and fully connected layers (matrix multiplication). The block sparsity we plan to deploy is not naturally recurring. 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 $8\times h\times k\times8$ block. For the matrix multiply $h = k = 1$, there is no difference from the previous discussions. We explain the training process.

3.1 Keras

We shall provide a repository using Keras [4] where we implements the contents of this section [6].

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 in another venue.

3.2 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 $\boldsymbol{W}_i \in \mathbb{R}^{8\times h\times w\times 8} \sim \mathbb{R}^{8\times8}$. 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 1, 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 data-set [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 1: 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). We use the sparsified models, we quantize them using Vitis AI, and we use them for time estimates (i.e., Section 6).

4 THE COMPILER AND ITS CODE GENERATION FOR AIE

We can 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. We use the tensor layout format BHWC, the tensors are represented in 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 can be tailored. The layout of the weight tensor is $COUT \times h \times w \times CIN$, which is similar to the caffe layout [13] and different from [8].

The main differences from our previous compilers are the parameterized representation of block sparsity and the capability to split tensors and computations accordingly. 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).

4.1 AIE Hardware Abstraction

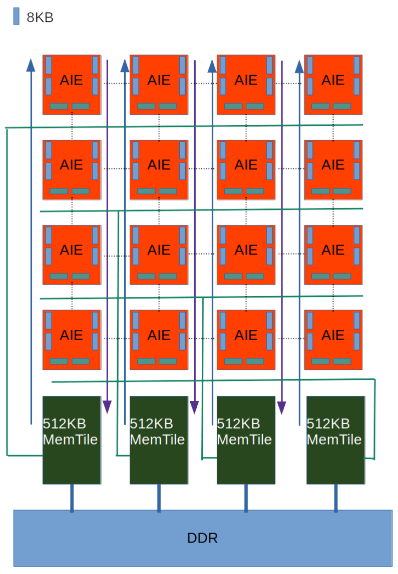


Figure 2: 4x4 AIE representation

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. 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). 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).

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.

4.2 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. As often as possible, we store the weights in the core and we reuse them (unless we need to *stream* the weight instead). The cores set is a cohort and we always choose symmetric computations. We do not merge two operations like convolution and max-pool and give three columns to convolution and one column to max-pool. Other solutions in the literature address this approach.

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).

The subvolume describes the smallest shape of the weights 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. The sparse data is smaller, we compute fewer operations, we can compute larger subproblems, and fewer data movements.

4.3 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. It means that an input tensor is distributed among the memtiles identified by one starting address and a final address and so do the weights. 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:

$$Y = Conv(\boldsymbol{X}, \boldsymbol{W}) = \dot{\sum}_{w} Conv(\boldsymbol{X}_w, \boldsymbol{W}) \tag{7}$$

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 of graph layers is a hard problem and often barely considered. We addressed it first (although by heuristics).

4.4 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:

$$Y_{height} = Conv(\boldsymbol{X}_{h=0}) \dot{+} \dot{\sum}_{h=1}^{9} Conv(\boldsymbol{X}_h) \dot{+} Conv(\boldsymbol{X}_{h=10}) \tag{8}$$

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 8, 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 8 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.

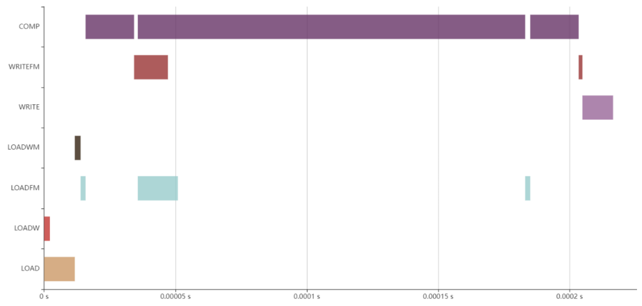


Figure 3: Resnet single convolution with padding for 4x4: legend AIE: LOAD activation from DDR to memtile, LOADW weights from DDR to memtile, LOADFM activation from memtile to AIE2 cores, LOADWM weights from memtile to AIE2, WRITE from memtile to DDR, WRITEFM from AIE2 to memtile, COMP Computation.

4.5 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.

Memtile are circular buffers so we can manage better the memory allocation; however, we do not allow streaming.

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 (and to the nearest neighbor). These communications are for activations (LOADFM). One memtile can broadcast to rows of cores (or the nearest neighbor), these are for weights (LOADWM). We assume that the column and row communications are parallel and each memtile core connection is 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. This is very optimistic. The execution time is a feature of the analysis. The estimate of the communications is more compelling and we can easily mute the computation contribution. Note however, that sparsity really reduces the computation 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.

4.6 Convolution example

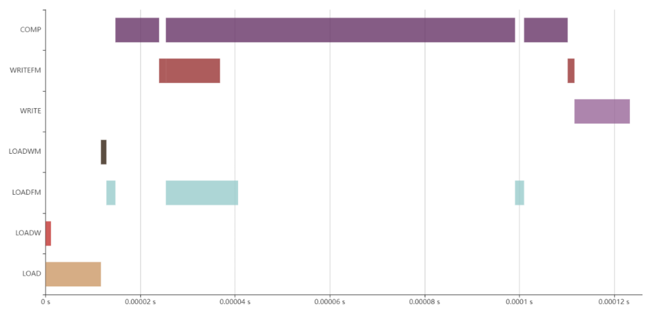


Figure 4: Resnet single convolution with padding and sparsity for 4x4 AIE

We present the time estimate for a convolution with padding, dense, and with 50% sparsity, see Figure 3 and 4. We explain in details how the time is estimated and our assumptions about how the hardware works. First, we load the weight and activations once in memtiles (LOAD and LOADW). There are actually one load per memtile for a total of four loads instructions per activation and weight. Because each load is to a different memtile, they are parallel. The activation and weight communications are using two different channels and they are parallel with 4 GBps bandwidth. There is a single load of the weights from memtiles to each core (LOADWM). This is done once and it is blocking. We can relax this condition.

There are three computations (COMP); that is, top padding, the body, and the bottom padding. For the top padding computation, there is the sequential execution of loads to cores (LOADFM), computation (COMP), and write to memtile (WRITEFM). This computation has iteration 1.

For the body computation, there are 9 iterations for three instructions: we can see the load, the computation, and the write are parallel. This is a simplification; there will be a little load poking out at the beginning and a writing poking out at the end. Then we conclude with padding at the bottom of the computation.

For this convolution, 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 only in space saving and may be spilling. In principle, now that we know we can relax sparsity requirements for those convolutions (and restart training).

5 DEPTH-WISE TILING

Take the subgraph schedule $L_0, L_1, \dots L_j$ and $u_j = 1 \times 1$ output of L_j subvector of T_j. A projection ¶ is a function taking u and projects the input sub-tensor needed to compute u. In reverse order, starting from L_j compute $¶(L_j, u_j)$ and propagate the projection. When a layer L_m and tensor T_m feeds two (or more) layers L_x and L_y. Then $u_m = \max(¶(L_x, u_x), ¶(L_y, u_y))$ when u_x and u_y are defined completely and *max* mean the largest. We carry the propagation of $¶(L_m, u_m)$ and we have $T_{-1} = ¶(L_0, u_0) = \prod_{l=j}^{0} ¶(L_l, u_l)$, for which there is no practical advantage. At the end of the propagation, imagine this is a generalized convolution with kernel size $k = ¶(L_0, u_0)$. Now if we repeat the process with $\dot{u}_j = 2 \times 2 \dots k + s = ¶(L_0, \dot{u}_0)$. 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.

6 RESULTS

In Table 2, 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 2. 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 is slower than 7x7 by using sparse. It may happen because a symmetric computation may use too small subvolume computations and thus more iterations for the same amount of data transfers. We will provide a full discussion if space permits.

6.1 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

Table 2: 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		failed	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

communications thanks to depth-wise tiling and the reduction of computation and weights communication by sparsity.

7 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.

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 [11]). 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.

This work stemmed from a collaboration between Xilinx and Numenta and the idea set forward by the authors in [1], where the models are too dense and there is a lot to improve.

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 [3, 12, 17]. Usually major attention is given only to matrix multiplication and GPUs [10] [15], we can work on only static sparsity at this time.

REFERENCES

[1] Subutai Ahmad and Luiz Scheinkman. 2019. How Can We Be So Dense? The Benefits of Using Highly Sparse Representations. arXiv:1903.11257 [cs.LG]

[2] AMD. 2020. rocSPARSE. https://rocsparse.readthedocs.io/en/master/.

[3] Jingwei Cai, Yuchen Wei, Zuotong Wu, Sen Peng, and Kaisheng Ma. 2023. Inter-layer Scheduling Space Definition and Exploration for Tiled Accelerators. *Proceedings of the 50th Annual International Symposium on Computer Architecture* (2023). https://api.semanticscholar.org/CorpusID:259177591

[4] François Chollet et al. 2015. Keras. https://keras.io.

[5] Paolo D'Alberto. 2013. https://github.com/paolodalberto/SparseFastMM/.

[6] Paolo D'Alberto. 2020. Block Sparsity and Training. https://github.com/paolodalberto/BlockSparsityyTraning.

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[8] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[9] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv:2210.17323 [cs.LG]

[10] Scott Gray, Alec Radford, and Diederik P. Kingma. 2017. GPU Kernels for Block-Sparse Weights. https://api.semanticscholar.org/CorpusID:52220661

[11] Benjamin Hawks, Javier M. Duarte, Nicholas J. Fraser, Alessandro Pappalardo, Nhan Tran, and Yaman Umuroglu. 2021. Ps and Qs: Quantization-aware pruning for efficient low latency neural network inference. *CoRR* abs/2102.11289 (2021). arXiv:2102.11289 https://arxiv.org/abs/2102.11289

[12] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), 554–566. https://api.semanticscholar.org/CorpusID:233740109

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (Orlando, Florida, USA) *(MM '14)*. Association for Computing Machinery, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[14] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Nir Shavit, and Dan Alistarh. 2020. Inducing and Exploiting Activation Sparsity for Fast Inference on Deep Neural Networks. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 5533–5543. https://proceedings.mlr.press/v119/kurtz20a.html

[15] Zhiyi Li, Douglas Orr, Valeriu Ohan, Godfrey Da costa, Tom Murray, Adam Sanders, Deniz Beker, and Dominic Masters. 2023. PopSparse: Accelerated block sparse matrix multiplication on IPU. arXiv:2303.16999 [cs.LG]

[16] NVIDIA. 2020. cuSPARSE. https://developer.nvidia.com/cusparse.

[17] Enrico Russo, Maurizio Palesi, Giuseppe Ascia, Davide Patti, Salvatore Monteleone, and Vincenzo Catania. 2023. Memory-Aware DNN Algorithm-Hardware Mapping via Integer Linear Programming. *Proceedings of the 20th ACM International Conference on Computing Frontiers* (2023). https://api.semanticscholar.org/CorpusID:260442361

Received TBD; revised TBD; accepted TBD