Weight Block Sparsity: Training, Compilers, and Accelerators

Taehee Jeong Akshay Jain Paolo D'Alberto

***Abstract*—Inference is a synonymous for performance: for example, more and faster classifications per seconds. As faster and tailor-made chip-lets are deployed in the field also larger and larger models are domineering the public's attention: quantization and sparsification are used to fit these large models into more reasonable sized ones. Quantization reduces the foot print per model weight and sparsification trims superfluous ones. 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 data compaction and computation splitting into threads and cores.**

Blocks present spatial locality that a vector operation can make full use and temporal locality for good reuse and cost amortization. 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 (Xilinx Versal FPGAs). We show application for general CPU, GPUS.

***Index Terms*—Sparsity, Performance, FPGA, Tools**

I. Introduction

We shall start by what we mean for block sparsity and for a vertical solution. This will provide a clear introduction to the subject. Block sparsity is an intuitive concept but it is also a little misunderstood because we all have a different mental picture of it. Take a matrix multiplication as 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 simply 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

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

Where the matrix $\boldsymbol{B}$ is a constant and diagonal and each submatrices $\boldsymbol{B}_2$ and $\boldsymbol{B}_1$ can split further down and show zero blocks of even smaller sizes. In this work, we chose the basic block of being of size $\boldsymbol{B}_i = 8 \times 8$. It is arbitrary but it is a great starting point for architectures based on AMD AIE2 products. For example,

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

Each block is a constant either non zero or zero:

$$\boldsymbol{B} = \begin{bmatrix} \gamma_0\boldsymbol{B}_0 & \cdots & \gamma_0\boldsymbol{B}_{n-1} \\ \gamma_n\boldsymbol{B}_n & \cdots & \gamma_{2n-1}\boldsymbol{B}_{2n-1} \\ \cdots & \cdots & \cdots \\ \gamma_{(n-1)n}\boldsymbol{B}_{(n-1)n} & \cdots & \gamma_{(n-1)^2}\boldsymbol{B}_{(n-1)^2} \end{bmatrix}$$

Of course, the matrix $\boldsymbol{A}$ can have any value and so $\boldsymbol{C}$. Some applications have some constraints about the row and columns of $\boldsymbol{B}_i$: for example, each row or column cannot be zero. In practice, we do not prune the network, we keep the same number of *channels* every where.

This is a well known data structure in the sparse computation field: We can use Compress Block Row or Column format (CBR). There are standard Matrix Sparse-Matrix Multiplication interfaces and algorithms for CPU and GPUs using this data format (where only one operand is sparse).

In the CBR format, the γ_i are not present and only non zeros elements are stored. In other architectures, we can choose to store all non zero blocks in row format and keep a matrix Γ of zeros and ones (or columns). The Γ is a bit matrix (here) but it can be represented as a short integer matrix representing the non-zero block column and in general two orders of magnitude $(8 \times 8 \times 4)$ smaller than the sparse or original $\boldsymbol{B}$ matrix. It is possible the problem has inherent block sparsity. In general, we are working with *man made* block sparsity: indeed for us, the granularity of the dense block is a property of the hardware.

In classification, the model weight size determines one important aspect of the model complexity: the number of operations per single output, the relation between layers (e.g., depth), and redundancy. The last property is important for resiliency and also to leave some space to learn new features if necessary. Exploiting sparsity is another way to reduce redundancies and computations. In our context, sparsity is the zeroing of weights (convolution and fully connected): we start with dense model using full precision and we have to find a way to chose the binary matrix Γ (which is also called a mask). For a matrix multiplication the mask is $\Gamma \in \mathbb{R}^{n \times m}$. In a convolution, n is the number of output channels (divided by 8) and m is the number of input channels (as above) and each $\boldsymbol{B}_i \in \mathbb{R}^{8 \times h \times w \times 8}$ where h and w are the height and width of the convolution kernel. Imagine $\boldsymbol{B}$ having a dimension going into the paper of size $k \times w$.

We explore training techniques (PyTorch, Keras is available in github): the most successful so far is the simplest: we take a pre-trained model, we compute a Γ per layer using a function to determine the blocks more likely to be zeros (Norm) and then we train the model till convergence or accuracy achieved. We take the sparse model and we quantize to 8-bit integer computation by using Vitis-AI quantizer. The final model is a XIR quantize model (assuming addition accumulators of 32 bits). See Section III.

For FPGA accelerator using AIE2, we have a custom compiler that takes the XIR model and an abstraction of a connected set of AIE2. See Section IV. For example, A DDR, one or two memory dedicated per column (512KB each called Mem-tile), 1 to 8 columns, 1 to 8 AIE2 cores per column, and each core has 8*8KB internal memory. There are vertical connections and there are horizontal connections. Given the HW and per layer, the compiler computes the maximum sub-volume computation per core. By heuristics and following a schedule, it computes a memory allocation in mem-tile for input, outputs, and weights. It formats the weights so that to exploit spatial distribution to Mem-tiles and cores into a compact form into a single tensor of weight, bias, and Γ.

With the schedule and the DDR-MemTile allocation, we generate all the explicit communications between DDR, MemTile, and cores. Knowing the subproblem sizes per core and the computation throughput and with a clear specification of what is executed in parallel: we can estimate the execution time per layer and of the entire network with an accuracy closer to a simulation. The code is valid, the native AIE2 compiler can interpret it, with all the DMA settings in place, and we tested for a single core in HW. We shall use this code to have a detailed time estimates for all parts of the computation: we shall show estimates for two CNN models, a few different AIE designs; see Section V.

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

II. Block-Sparse Matrix-Matrix Multiplication

As mental and practical exercise, consider Γ and Ω two appropriate 0,1 matrices so that for square matrices in $\mathbb{R}^{N\times N}$

$$\boldsymbol{C} = (\Gamma \boldsymbol{A}) * (\Omega \boldsymbol{B}) \tag{5}$$

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

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

Thanks to the sparsity and if we store only non-zeros, then $gamma_{i*N+k}$ is at the very least contiguous but ω_{k*N+j} and the right operand accesses are far from being neither simple not contiguous.

$$\dot{\Omega}\dot{\boldsymbol{B}} = (\Omega \boldsymbol{B})^t = \Omega^t \boldsymbol{B}^t \tag{7}$$

Although expensive, the transposition of a sparse matrix is a sorting algorithm: We start from a row order to a column order, then consider K non-zeros, $K \log_2(K)$.

$$\boldsymbol{C}_{i*N+j} = \sum_k (\gamma_{i*N+k}\boldsymbol{A}_{i*N+k})(\dot{\omega}_{j*N+k}\dot{\boldsymbol{B}}_{j*N+k}) \tag{8}$$

There will be a meaningful product to compute if and only if $\gamma_{i*N+k} = 1$, $\dot{\omega}_{j*N+k} = 1$, and $i*N+k = j*N+k$. Without any extra information, this is simply merge sort. If we take $\gamma_{i*N+k} = i*N+k$ when the relative block is non zero, nothing otherwise, create a vector, and do the same for $\dot{\omega}_{j*N+k}$, then we merge-sort these vectors, we do computations only on equality: we have to inspect each non zero elements M_0 and M_1, which is $\leq O(\frac{N}{K})$. If you like to break codes yourself, see how the Sparse Sparse Matrix multiplication using Coordinate Block Structure (COO) is, please go play with [1]. We have a parallel sorting and a parallel matrix multiplication.

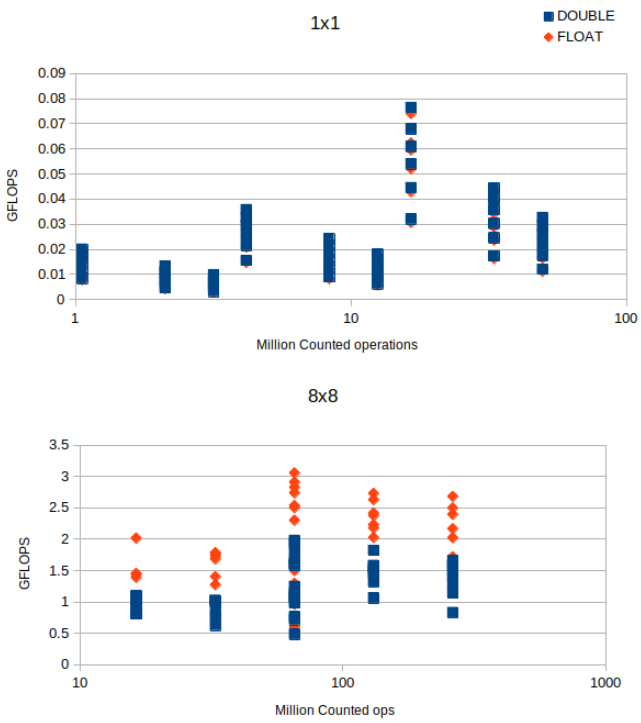

Fig. 1. Block 1x1 and 8x8 performance

Now, the intuitive part, 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 number of blocks per dimension and thus the overhead for sparsity and sorting, is basically $\frac{1}{2}\frac{N}{k}$. Larger the k smaller the overhead. The relative performance of the k^3 multiplication is better as k get larger because spatial and temporal locality and optimized code for a constant/parameterized k.

In Figure 1, we present two scatter plots: on the abscissa number of effective multiplication and addition, on the ordinate the performance in GFLOPS, when the sparsity dense block is 1x1 and 8x8. Given the same problem, we may use more threads and thus the Jenga of points. We can see that given he same number of effective operations, the block permits better performance and exploits better performance for each precisions: see a $2\times$ performance for single precision computation versus double precision. This is a sparse computation and thus the poor peformance (in GFLOPS) is actually expected (the peak performance in this architecture is about 500+GFLOPS).

III. 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 are seeking to deploy is not naturally recurring and it has to be made. We must train the network so that we can zero blocks of data.

First, let us clarify block sparsity for convolution weights, then we clarify our training process. A convolution has a weight tensor in four dimension: $\boldsymbol{W} \in \mathbb{R}^{c_{out} \times h \times k \times c_{in}}$. If you can visualize the h and k dimension going into the paper: 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 $h = k = 1$ and there is no difference from the previous discussions.

A. Keras

We provide a repository using Keras [2] where we implements the contents of this section: Any one can reproduce and break [3]. We target convolutions only and without quantization. The idea of the framework 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 [4]. The goal is to choose Γ so that we achieve the required sparsity and to have the minimum loss in accuracy. A little notation first:

We start from an optimized network and assume a loss function $\ell(x, w)$. In a close interval of the optimal solution $\boldsymbol{w}_0$ we have:

$$\ell(x, \boldsymbol{w}_0 + \triangle\boldsymbol{w}) = \ell(x, \boldsymbol{w}_0) + \nabla\ell * \triangle\boldsymbol{w} + (\triangle\boldsymbol{w})^t * H * (\triangle\boldsymbol{w}) + \epsilon \tag{9}$$

Where the gradient of the loss function is about zero and defined as

$$\nabla\ell * \triangle\boldsymbol{w} = \sum_{w_i} \frac{\partial\ell}{\partial w_i} \triangle w_i \to 0 \tag{10}$$

The second component is the Hessian, it is symmetric and either definite positive or negative as a function of the loss.

$$(\triangle\boldsymbol{w})^t * H * (\triangle\boldsymbol{w}) = \sum_{w_i} \triangle w_i \sum_{w_j} \frac{\partial\ell}{\partial w_i \partial w_j} \triangle w_j \tag{11}$$

We start from $\boldsymbol{w}_0$, the current optimal weight, and we must choose how to reduce to zeros the weight and which ones.

In practice, using the code available in [3], we started with a accuracy of 75% top-1 and we ended up with a 68% without accounting for quantization. The investigation using PyTorch, sparsity and quantization achieves only 3% top-1 accuracy drop (instead of 6+), making the training for sparsity quite worth while (Section III-G).

B. Γ chosen once and full training ahead

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

We can then sort them in ascending order and for example choose the first 50% to set to zero. Then we start re-training. We do this for the whole network or for one convolution at a time. This is the approach used in Section III-G

C. Γ chosen in steps and small fine tuning

Let say that for every convolution, $n_i = \sum \gamma_i$, which is the number of blocks. We would like to reduce this number to $\frac{n_i}{2}$. For each epoch (say every two training epochs), we consider the current non-set-yet mask elements $\sum \gamma_i = k < \frac{n_i}{2}$. We compute our importance measure for all in ascending order. This time, we zero the first $\min(\frac{5}{100}n_i, 1)$. We keep this process until we reach 50% sparsity. At each iteration at least one block will be set to zero.

D. Γ trainable as optimization problem

We think it is worth mentioning: If we want to make Γ part of the optimization process as trainable variable we could introduce as a penalty function into the loss $\ell(x, w) + \lambda(w)$.

First let us introduce an approximation for the $\max(x)$, so when in this section you will read $\max, \min$, this is a log sum exponetial, which is continuous, derivable everywhere, and convex:

$$\max(\boldsymbol{x}) = LSE(\boldsymbol{x}, \alpha) = \frac{1}{\alpha} \log \sum e^{x_i * \alpha} \tag{12}$$

With T we represent the number of non zero block in Γ

$$\begin{aligned}\lambda = &- (\max(\Gamma) - \min(\Gamma)) \\ &+ \beta * L2(\Gamma - T) + \iota * L1(\Gamma)\end{aligned} \tag{13}$$

This is a simplified loss so that we minimize the valu of Γ but also we try to maximize the difference of the elements.

$$\begin{aligned}\lambda = &\max(-\Gamma, 0) + \max(\Gamma - 1, 0) - (\max(\Gamma) - \min(\Gamma)) \\ &+ \beta * L2(\Gamma - T) + \iota * L1(\Gamma)\end{aligned} \tag{14}$$

This last penalty function represents our attempt to state the γ_i should be positive and in the interval $[0, 1]$ and in a such a way that we maximize the distance of the elements between 0s and 1s.

$$\begin{aligned}\lambda = &\max(-\Gamma, 0) + \max(\Gamma - 1, 0) - \frac{\min(\Gamma)}{\max(\Gamma)} \\ &+ \beta * L2(\Gamma - T) + \iota * L1(\Gamma)\end{aligned} \tag{15}$$

We can exploit this functionality in Keras: the penalty function can be added quite easily and per convolution (if

you like) and it is available in the code reference. We could not use it successfully: we do not know if this is because the lack of the penalty it self, the optimizer, or the lost function, or just our own fault.

E. Hessian and Fisher Information

If we have N parameters/weights, the Hessian $H \in R^{N \times N}$ has quite the space complexity (consider even small models could have million parameters). When we are already close to the optimal solution or we are trying to move from the optimal solution without using information from the gradient, the Hessian provides the most information close by an already established solution point. There are also way to compute the Hessian and the effect of the hessian by using Fisher information [5]–[7] and this will reduce to the computation of the gradient of the loss function and the citation within. We could not reproduce the authors results.

The goal is to use the H so that we know in advance what will be the effect of zeroing weights. This will boil down to a $O(h_i * w_i)$ that we can use to estimate the importance of a block

F. Diagonal Hessian

We applied a Fisher measure and we just computed $\nabla^2 \ell$, that is we compute just the diagonal of the Hessian. Again, we use the L_2 over the normalized weight and went thought the process of training. The elements of the diagonal are not representative in general, but they are a good approximation of the contribution of a single weight. Clearly, convolution is linear and any CNN is a deep network.

The Fisher and $\nabla^2 \ell$ did not provide any main advantages. But this information is found very useful in quantization and optimizations within the same field and application. We embrace the community to help us to understand this better.

G. PyTorch

In Table I, we show the results by using one-time masks and full training: VGG-16, ResNet-50, Inceptionv3 on ImageNet20 (20 classes) and ImageNet1k (1000 classes). See Section III-B. 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 [4] 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 sample for re-training.

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 compared to block of 8×8. Fine-grained sparsity (1×1 block) does not sacrifice any accuracy (i.e., almost any). We use the sparsified Resnet50, we quantized using Vitis AI, and we shall use it for estimate the time advantages (i.e., Section V).

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

IV. COMPILER AND CODE GENERATION AIE

Within our Xilinx/AMD effort, we can take a PyTorch/Keras, quantize it using Vitis AI, and create an intermediate representation by using what we call Xilinx Intermediate Representation (XIR). This is a graph of operations that read and write tensors. A convolution has one quantized input: we use the 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 as the input and with a proper scale. The weights and bias are properties of the convolutions, as such they can be tailored.

For this project, we are at the third generation of a compiler for custom architectures (previously DPUV1 and DPUV3int8 [8], [9]). The main common harwdware feature for this new architecture is the availability of two main memory levels: DDR and MemTile, and for such memory we import the philosophy of memory data layout and memory allocation (of the previous compilers).

All weights are statically prepared into DDR and we move them explicitly towards the inner levels. Inputs and outputs have designated space DDR (to communicate externally with PCI connection or RAM). DDR can and it will be used for swap tensors in case the inner levels do not allow allocation. The compiler can explore different schedules: the memory allocation to Mem-Tile is basically coloring algorithm plus some heuristics. In this architecture, we do not allow *streaming* of neither data or weights (because they share space in Mem-Tile). In previous architecture, we could.

A. AIE Hardware abstraction

For this presentation, see Figure 2, we work with a mesh of 4x4 AIE2 cores connected by 4 horizontal and 4 vertical interconnections. Each core has 8 banks memories for a total 64KB. About 6 banks are used and input/output/weight buffers and 2 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).

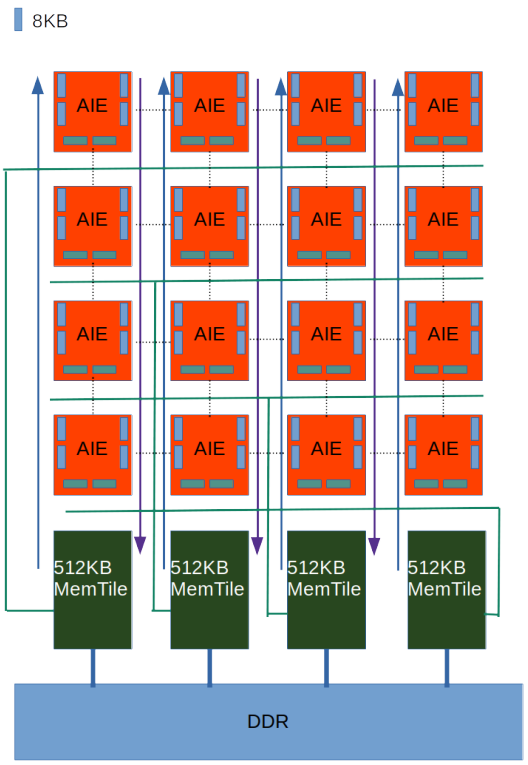

Fig. 2. 4x4 AIE representation

There are four mem-tiles: each 512KB and each can either be connected to one columns and its direct neighbor column, or it can be connected to a row and its neighbor. The total amount of space is 2MB.

A Memtile can broad-cast data per column or per row. We can dedicate a memtile for weights, one for activations, or we can share. To maximize the computation parallelism, the core will write data per column into memtile: there are different designs that can be used. We shall explore the case where Inputs/Outputs/Weights are evenly distributed across mem-tiles although the distribution is different per tensor.

The DDR is connected with two channels to write into each mem-tile and each mem-tile can use two channels to write into DDR. DDR and Mem-tile communications are parallel. The abstraction can be extended to more complex connections (from 1×1, 8×2, to 8×8 and more rows of memtiles designed for larger chips). But 4×4 is representative for the AIE engines you will find in the next generation of CPU+FPGA chips.

B. Sub Volume, Data compression and Data Movement

There are a few assumptions. The computation is split by column: each output tensor is split evenly by width, so each memtile will store different columns by width, each core will compute different output channels, and the computation streams computing the height of the tensor by using core input and output ping/pong. As often as possible, we store the weights in the core and we reuse them as much as possible (unless we need to stream the weight instead). The set of cores is a cohort and symmetric computations are always selected. This means that we do not merge two operations like convolution and max-pool and give three columns to convolution and one column to max-pool. Other approach in the literature will do this more advanced methodology.

Before we start describing the memory allocation, we have to explain and compute the size of the sub-volume computation at AIE core. That is, 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 at least one output channel and one row 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 spils) and we can manage to split the input channels (this will require to split the weights accordingly and accumulate). We call W-Split the distribution of tensor by columns in the AIE mesh. We can CIN-split when we need to split by input channel. The split of the computation is also called tiling. Once we have the subvolume, we know if input and output tensor need W-split or CIN-split. This in general, means that the tensor cannot fit completely in Mem-tile and part of it will be moved. Or the computation will need to read from mem-tile the same inputs or weights multiple times.

At this stage, the subvolume describes the smallest shape of the weights we need to manage. We compress the weight accordingly to such a subvolume so that any data movement will always be a multiple of the subvolume and can be a single load. Such a compress data will have the same properties whether is sparse or not. Of course, sparse data is smaller and we compute fewer operations.

C. Schedule and memory allocation

This is a two level memory hierarchy. During the scheduling of each layer, we evaluate what can fit in mem tile. Here, Activations and weights share the space. At each step the memory allocation will check if we can allocate a tensor in memtile. If we cannot, we evict all tensors into DDR and then split/time the computation. If we can, we do, and we will evict the tensors no longer needed accordingly.

At the end of this stage, every tensor will be associated to an address in memtile or DDR (or both). If there are only DDR addresses, the compiler will take the basic computation and, by an heuristics, will split the computation (thus the output tensor) by width, output channel, height, and input channel (non necessarily in this order). Every computation will have the DDR to and from MemTile and its data movements for the first two levels of the memory hierarchy.

We use a implementation that takes a recursive and iterative approach in tiling [9]: For clarity, $\sum$ is a sum or reduction and $\dot{\sum}$ is a parallel loop and a W-split can be written as

$$\boldsymbol{Y} = Conv(\boldsymbol{X}, \boldsymbol{W}) = \dot{\sum_{w=1}^{k}} Conv(\boldsymbol{B}_w, \boldsymbol{W}) \qquad (16)$$

The split is pre-computed as function of the foot print, before and after each convolution there will be an explicit data movement. At this stage each input, output, and weights will have an address associated with each sub-computation. Then

the code generation of each $Conv(\boldsymbol{B}_w, \boldsymbol{W})$ is independent and recursive and as needed there will be specific split of the computation accordingly. This could be simply described using loops but it is actually a tree (i.e, a root with k children). If the convolution has strides and large kernel, each sub-convolution may have overlap data but defined addresses and data movement if necessary. For example, computing the output by rows and the weights are reused.

$$\forall w,\ Conv(\boldsymbol{B}_w, \boldsymbol{W}) \rightarrow \sum_{h=1}^{\cdot}{}^{\ell} Conv(\boldsymbol{B}_{w,h}, \boldsymbol{W}) \qquad (17)$$

D. Code Generation

The compiler recursively create a list of operations smaller and smaller that can actually be executed Mem-Tile to Mem-Tile. In practice, there is a further decomposition using only AIE cores but it is completely determined by the subvolume computation we did previously. We can explicitly determine the data movements from mem-tile to core and back.

A peculiar feature of the current system is the concept of *iteration*. Using locks and chaining them (locks like in semaphores)

We can create a single iteration instruction from the prospective of a single core (as a SIMD instruction) but driving all cores at once (ASM code)

```
LOAD Lock k_0 mem-tile add core add iter i
CONV iteration      i
SAVE Lock k_1 mem-tile add core add iter i
```

These three operations will be execute a number of iterations *i* and using a ping/pong they will load different slices of data and compute different slices of data. The key ingredient for this to work is the volumes of input and output tensors are all the same.

In our environment padding is common and we can manage by a custom load (from memtile to core) and this requires a custom load that will not be repeated

```
## Head top padding
LOAD Lock k_0 mem-tile add_0 core add iter 1
CONV iteration 1
SAVE Lock k_1 mem-tile add_1 core add iter 1
## Body iteration
LOAD Lock k_2 mem-tile add_2 core add iter i
CONV iteration i
SAVE Lock k_3 mem-tile add_3 core add iter i
## tail
LOAD Lock k_4 mem-tile add_4 core add iter 1
CONV iteration 1
SAVE Lock k_5 mem-tile add_5 core add iter 1
```

At this stage we have all the information: the code generation per layer is a two pass process: first, we generate code for the all load/store and then we combine them into chain having dependency so that to be logically correct and as fast as possible. Take the code above and we introduce a chain information.

E. Time Estimation

During and especially once we generated the code for all data movements, data volumes and destinations, the trigger of the computation, and the sub volume per computation we can estimate quite accurately the execution time for each operations and account their parallel execution.

V. Results

VI. Conclusions

References

[1] P. D'Alberto, https://github.com/paolodalberto/SparseFastMM/, 2013.

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

[3] P. D'Alberto, "Block sparsity and training," https://github.com/paolodalberto/BlockSparsityyTraning, 2020.

[4] 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.

[5] Z. Yao, A. Gholami, S. Shen, K. Keutzer, and M. W. Mahoney, "Adahessian: An adaptive second order optimizer for machine learning," *AAAI (Accepted)*, 2021.

[6] S. Yu, Z. Yao, A. Gholami, Z. Dong, M. W. Mahoney, and K. Keutzer, "Hessian-aware pruning and optimal neural implant," *CoRR*, vol. abs/2101.08940, 2021. [Online]. Available: https://arxiv.org/abs/2101.08940

[7] B. Zandonati, A. A. Pol, M. Pierini, O. Sirkin, and T. Kopetz, "Fit: A metric for model sensitivity," 2022.

[8] P. D'Alberto, V. Wu, A. Ng, R. Nimaiyar, E. Delaye, and A. Sirasao, "Xdnn: Inference for deep convolutional neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 2, jan 2022. [Online]. Available: https://doi.org/10.1145/3473334

[9] P. D'Alberto, J. Ma, J. Li, Y. Hu, M. Bollavaram, and S. Fang, "DPUV3INT8: A compiler view to programmable FPGA inference engines," *CoRR*, vol. abs/2110.04327, 2021. [Online]. Available: https://arxiv.org/abs/2110.04327

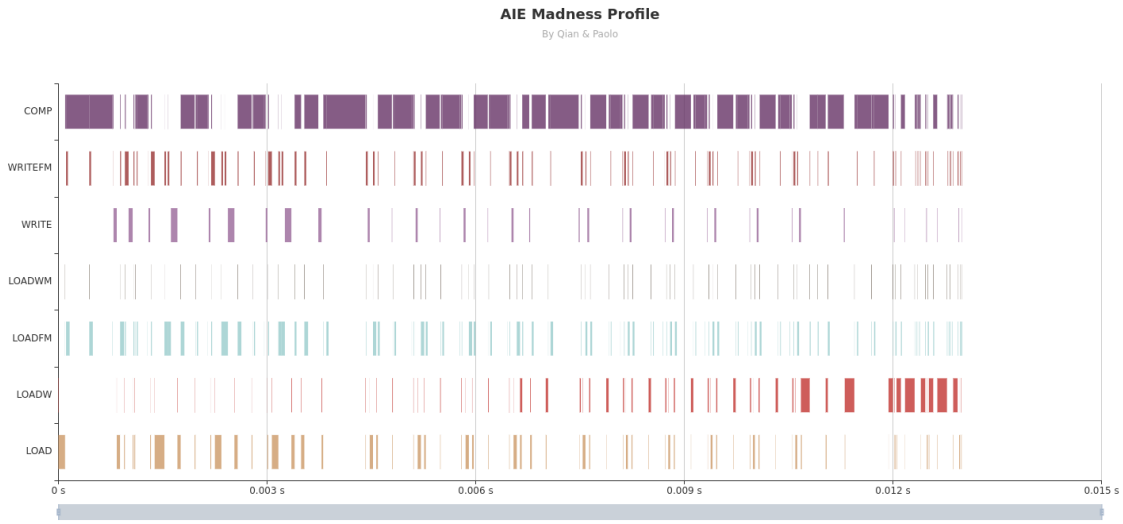

Fig. 3. Resnet50 for 4x4 AIE with dense weights

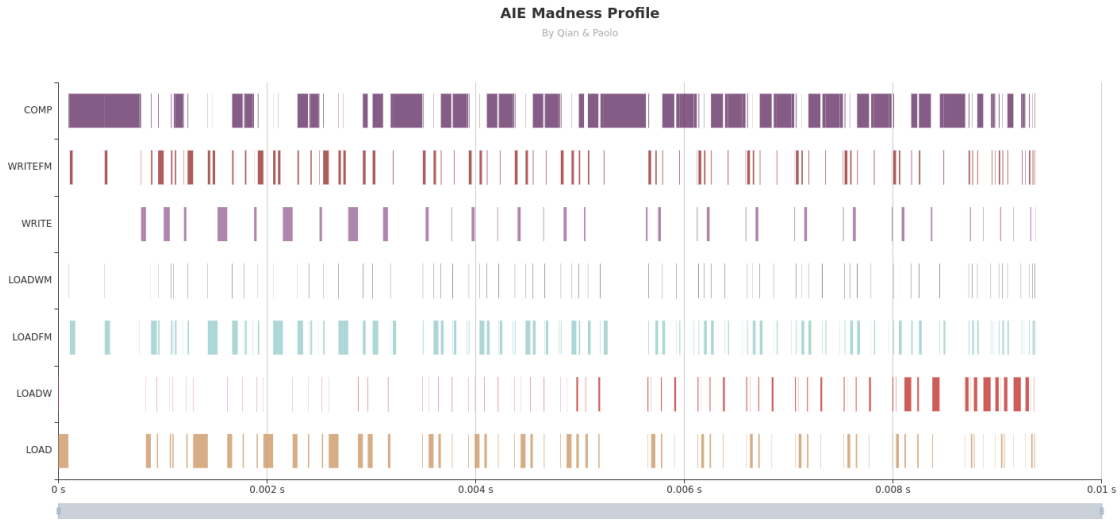

Fig. 4. Resnet50 for 4x4 AIE with 50% sparse weights

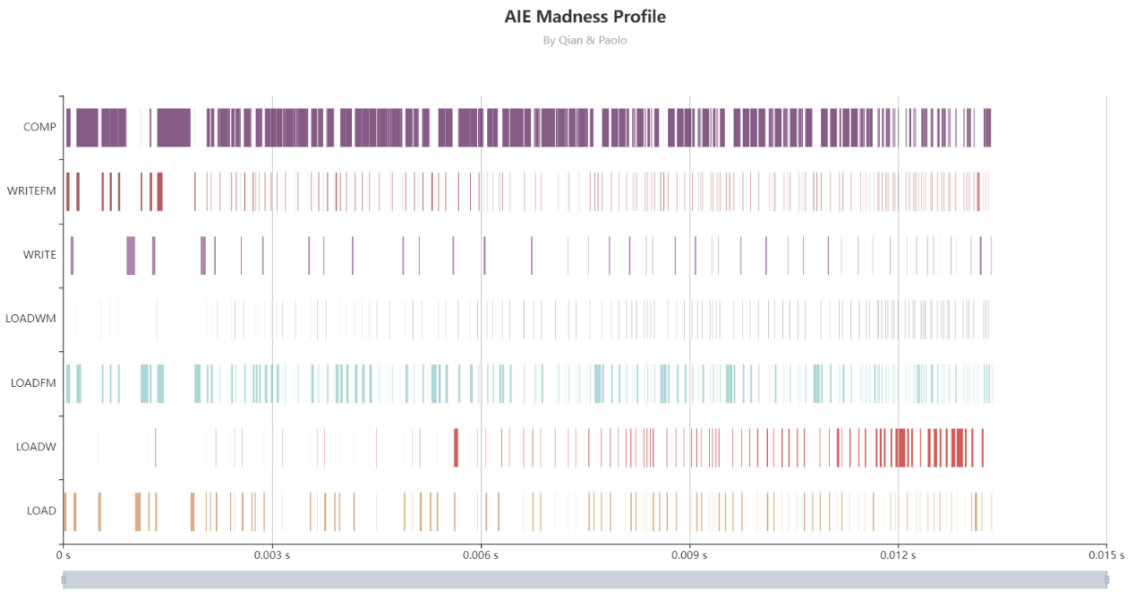

Fig. 5. Inception V3 for 4x4 AIE with dense weights

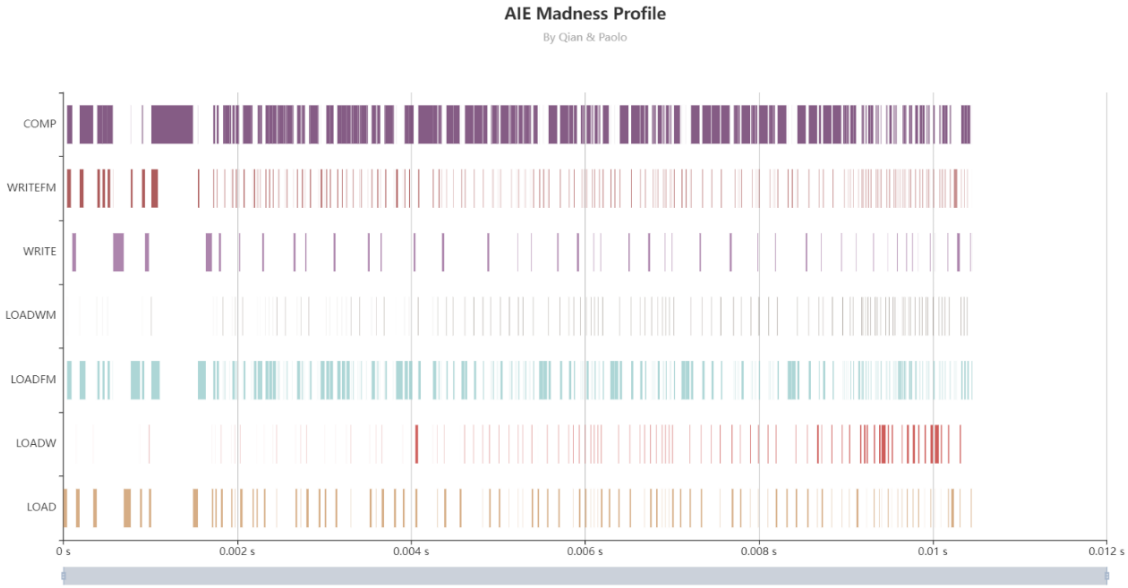

Fig. 6. Inception V3 for 4x4 AIE with 50% sparse weights