CapsAcc: An Efficient Hardware Accelerator for CapsuleNets with Data Reuse

Alberto Marchisio, Muhammad Abdullah Hanif, Muhammad Shafique
Vienna University of Technology (TU Wien), Vienna, Austria
{alberto.marchisio, muhammad.hanif, muhammad.shafique}@tuwien.ac.at

***Abstract*—Recently, *CapsuleNets* have overtaken traditional Deep Convolutional Neural Networks (CNNs), because of their improved generalization ability due to the multi-dimensional capsules, in contrast to the single-dimensional neurons. Consequently, CapsuleNets also require extremely intense matrix computations, making it a gigantic challenge to achieve high performance. In this paper, we propose *CapsAcc*, the first specialized CMOS-based hardware architecture to perform CapsuleNets inference with high performance and energy efficiency. State-of-the-art convolutional CNN accelerators would not work efficiently for CapsuleNets, as their designs do not account for unique processing nature of CapsuleNets involving multi-dimensional matrix processing, squashing and dynamic routing. Our architecture exploits the massive parallelism by flexibly feeding the data to a specialized systolic array according to the operations required in different layers. It also avoids extensive load and store operations on the on-chip memory, by reusing the data when possible. We synthesized the complete CapsAcc architecture in a 32nm CMOS technology using Synopsys design tools, and evaluated it for the MNIST benchmark (as also done by the original CapsuleNet paper) to ensure consistent and fair comparisons. This work enables highly-efficient CapsuleNets inference on embedded platforms.**

I. Introduction

Machine Learning (ML) algorithms (especially Deep Convolutional Neural Networks, CNNs) are widely used for Internet of Things and Artificial Intelligence applications, such as computer vision [6] and speech recognition [2]. CapsuleNets [12] [4] supersede traditional CNNs in multiple tasks (like image classification) [12], by encapsulating multi-dimensional features across the layers. The CapsuleNets are deeper in width than in height, when compared to CNNs, as each capsule incorporates the information hierarchically, thus preserving other features like position, orientation and scaling (see an overview of CapsuleNets in Section II).

State-of-the-art CNN accelerators [3] [1] [5] [11] [8] proposed energy-aware solutions for inference using traditional CNNs. **We are the first to propose a hardware accelerator-based architecture for the complete CapsuleNets inference**. The existing CNN accelerators cannot compute several key operations of the CapsuleNets (i.e., multi-dimensinal matrix processing, squashing activation function, and routing-by-agreement algorithm to propagate data towards the otuput) with high performance. A direct feedback connection from the outputs coming from the activation unit back to the inputs of the processing element allows to reuse data efficiently. Thus, such key optimizations can highly increase the performance and reduce the memory accesses.

Our Novel Contributions:

1) We analyze the memory requirements and the performance in the forward pass of CapsuleNets, through experiments on a GPU, which allows to identify the corresponding bottlenecks (see Section III).
2) We propose *CapsAcc* (see Section IV), an accelerator that enables performance and energy-efficient inference on CapsuleNets with specialized hardware accelerators for their functional blocks and multi-dimensinal matrix processing with massively parallel multiply-and-accumulate (MAC) arrays, as well as an efficient data reuse based mapping policy.
3) We implement and synthesize the complete CapsAcc architecture for a 32nm technology using the ASIC design flow, and perform evaluations for performance, area and power consumption. We performed the functional and timing validation through gate-level simulations (see Section V). Our results demonstrate a speed-up of 12× in the ClassCaps layer, of 172× in the Squashing and of 6× in the overall CapsuleNet inference, compared to a highly optimized GPU implementation.

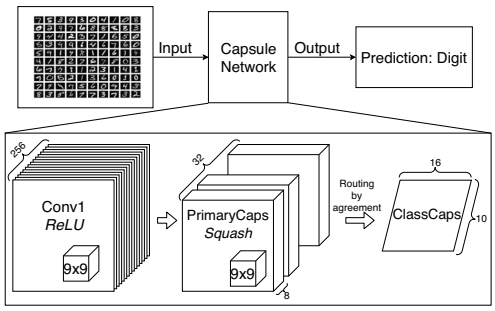

Fig. 1: An overview of the CapsuleNet architecture, based on the design of [12] for the MNIST dataset.

II. Background: An Overview of CapsuleNets

Since we analyze the inference process in this paper, the layers and the algorithms involved in the training process *only* (e.g., decoder, margin loss and reconstruction loss) are not discussed.

A. CapsuleNet Architecture

Figure 1 illustrates the CapsuleNet architecture [12] designed for the MNIST [7] dataset. It consists of 3 layers:

- **Conv1:** traditional convolutional layer, with 256 channels, with filter size of 9x9, stride=1, and ReLU activations.
- **PrimaryCaps:** first capsule layer, with 32 channels. Each eight-dimensional (8D) capsule has 9x9 convolutional filters with stride=2.
- **ClassCaps:** last capsule layer, with 16D capsules for each output class.

At the first glance, it is evident that a capsule layer contains multi-dimensional capsules, which are groups of neurons nested inside a layer. One of the main advantages of CapsuleNets over traditional CNNs is the ability to learn the hierarchy between layers, because *each capsule element is able to learn different types of information* (e.g., position, orientation and scaling). Indeed, CNNs have limited model capabilities, which they try to compensate by increasing the amount of training data (with more samples and/or data augmentation) and by applying pooling to select the most important information that will be propagated to the following layers. In capsule layers, however, the outputs are propagated towards the following layers in form of a prediction vector, whose size is defined by the capsule dimension. A simple visualization of how a CapsuleNet works is presented in Figure 2. After the weight matrix multiplication W_{ij}, the values $\hat{u}_{i|j}$ are multiplied by the coupling coefficients c_{ij}, before summing together the contributions and applying the squash function. The coupling coefficients are computed and

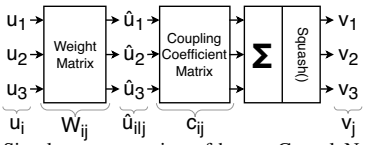

Fig. 2: Simple representation of how a CapsuleNet works.

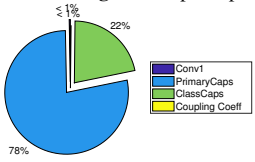

Fig. 3: Trainable parameters across different layers.

	Inputs	# parameters	Outputs
Conv1	784	20992	102400
PrimaryCaps	102400	5308672	102400
ClassCaps	102400	1474560	160
Coupling Coeff	160	11520	160

TABLE I: Input size, number of trainable parameters and output size of each layer of the CapsuleNet.

updated at run-time during each inference pass, using the *routing-by-agreement* algorithm (Section II-C).

B. Squashing

It introduces the nonlinearity into an array and normalizes the outputs to values between 0 and 1. Given s_j as the input of the capsule j (or, from another perspective, the sum of the weighted prediction vector) and v_j as its respective output, the squashing function is defined by the Equation (1).

$$v_j = \frac{||s_j||^2}{1 + ||s_j||^2} \frac{s_j}{||s_j||} \quad (1)$$

C. Routing-by-Agreement Algorithm

The predictions are communicated between two consecutive layers through the routing-by-agreement algorithm. It introduces a loop in the forward (inference) pass, because the coupling coefficients c_{ij} are learned during routing, as their values depend on the current data. Hence, this step can cause a computational bottleneck, as demonstrated in Section III.

III. Motivational Analysis of CapsuleNet Complexity

In the following, we perform a comprehensive analysis to identify how CapsuleNet inference is performed on a standard GPU platform, like the one used in our experiments, i.e., the Nvidia Ge-Force GTX1070 GPU (see Figure 4). First, in Section III-A we quantitatively analyze how many trainable parameters per layer must be fed from the memory. Then, in Section III-B we benchmark our pyTorch [13] based CapsuleNet implementation for the MNIST dataset to measure the performance of the inference process on our GPU.

A. Trainable parameters of the CapsuleNet

Figure 3 shows quantitatively how many parameters are needed for each layer. As evident, the majority of the weights belong to the PrimaryCaps layer, due to its 256 channels and 8D capsules. Even if the ClassCaps layer has fully-connected behavior, it counts just for less than 25% of the total parameters of the CapsuleNet. Finally, Conv1 and the coupling coefficients counts for a very small percentage of the parameters. The detailed computation of the parameters is reported in Table I. Based on that, we make an observation valuable for designing our hardware accelerator: *by considering an 8-bit fixed point weight representation, we can estimate that an on-chip memory size of 8MB is large enough to contain every parameter of the CapsuleNet.*

B. Performance Analysis on a GPU

At this stage, we evaluate the time required for an inference pass on the GPU. The experimental setup is shown in Figure 5. Figure 6 shows the measurements for each layer. The ClassCaps layer is the computational bottleneck, because it is around 10× slower than the previous layers. To obtain more

Fig. 4: Nvidia Ge-Force GTX1070.

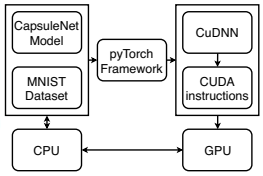

Fig. 5: GPU analyses setup.

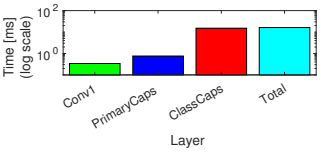

Fig. 6: Layer-wise performance of the inference pass of the CapsuleNet.

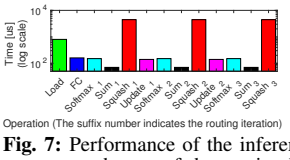

Fig. 7: Performance of the inference pass on each step of the routing-by-agreement algorithm.

detailed results, a further analysis has been performed, regarding the performance for each step of the routing-by-agreement (Figure 7). It is evident that *the Squashing operation inside the ClassCaps layer represents the most compute-intensive operation*. This analysis gives us the motivation to *spend more effort in optimizing routing-by-agreement and squashing* in our CapsuleNet accelerator.

C. Summary of Key Observations from our Analyses

From the analyses performed in Sections III-A and III-B, we derive the following key observations:

- The CapsuleNet inference performed on GPU is more compute-intensive than memory-intensive, because the bottleneck is represented by the squashing operation.
- A massive parallel computation capability in the hardware accelerator is desirable to achieve the same or a better performance than the GPU for Conv1 and ClassCaps layers.
- Since the overall memory required to store all the weights is quite high, the buffers located in between the on-chip memory and the processing elements are beneficial to maintain high throughput and to mitigate the latency due to on-chip memory reads.

IV. Designing the CapsAcc Architecture

Following the above observations, we designed the complete CapsAcc accelerator and implemented it in hardware (RTL). The top-level architecture is shown in Figure 8, where the blue-colored blocks highlight our novel contributions over other existing accelerators for CNNs. The detailed architectures of different components of our accelerator are shown in Figure 9. Our CapsAcc architecture has a systolic array supporting computational parallelism for multi-dimensional matrix operations. The partial sums are stored and properly added together by the accumulator unit. The activation unit performs different activation functions, according to the requirements for each stage. The buffers (Data, Routing and Weight Buffers) are essential to temporarily store the information to feed the systolic array without accessing every time the data and weight memories. The two multiplexers in front of the systolic array introduce the flexibility to process new data or reuse them, according to the operations (layers and routing-by-agreement operations) under process:

- Conv1 and PrimaryCaps: the weights are held inside the processing elements of the systolic arrays to exploit filter reuse across different input data.
- Sum generation and squash, update and softmax: the horizontal feedback channel allows to reuse the predictions $\hat{u}_{j|i}$, while the routing buffer is utilized to store either the coupling coefficients c_{ij} or the outputs v_j.

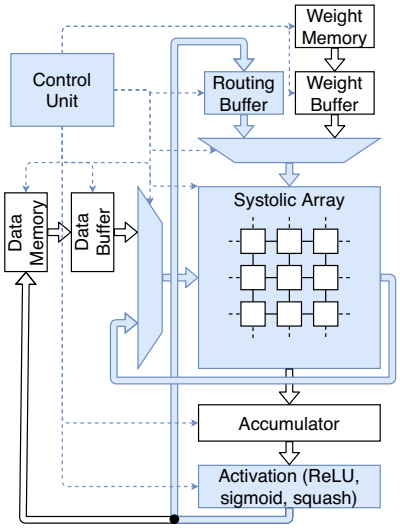

Fig. 8: Overview of our CapsAcc Architecture.

A. Systolic Array

The systolic array of our CapsAcc architecture is shown in Figure 9a. It is composed of a 2D array of Processing Elements (PEs), with n rows and m columns, similarily to [5]. For illustration and space reasons, Figure 9a presents the 4×4 version, while in our actual CapsAcc design we use a 16×16 systolic array. The inputs are propagated towards the outputs of the systolic array both horizontally (Data) and vertically (Weight, Partial sum). In the first row, the inputs corresponding to the Partial sums are zero-valued, because each sum at this stage is equal to 0. Meanwhile, the Weight outputs in the last row are not connected, because they are not used in the following stages.

Figure 9b shows the data path of a single Processing Element (PE). It has 3 inputs and 3 outputs: Data, Weight and Partial sum, respectively. The core of the PE is composed of the sequence of a multiplier and an adder. As shown in Figure 9b, it has 4 internal registers: (1) *Data Reg.* to store and synchronize the Data value coming from the left; (2) *Sum Reg.* to store the Partial sum before sending it to the neighbor PE below; (3) *Weight$_1$ Reg.* synchronizes the vertical transfer; (4) *Weight$_2$ Reg.* stores the value for data reuse. The latter is particularly useful for convolutional layers, where the same weight of the filter must be convolved across different data. For fully-connected computations, the second weight register introduces just one clock cycle latency, without affecting the throughput. The bit-widths of each element have been designed as follows: (1) each PE computes the product between an 8-bit fixed-point Data and an 8-bit fixed-point Weight; and (2) the sum is designed as a 25-bit fixed-point value. At full throttle, each PE produces one output-per-clock cycle, which also implies one output-per-clock cycle for every column of the systolic array.

B. Accumulator

The Accumulator unit consists of a FIFO buffer to store the Partial sums coming from the systolic array, and sum them together when needed. The multiplexer allows the choice to feed the buffer with the data coming from the systolic array or with the one coming from the internal adder of the Accumulator. We designed the Accumulator to have 25-bit fixed-point data. Figure 9c shows the data path of our Accumulator. In the overall CapsAcc there are as many Accumulators as the number of columns of the systolic array.

C. Activation Unit

The Activation Unit follows the Accumulators. As shown in Figure 9d, it performs different functions in parallel, while the multiplexer (placed at the bottom of the figure) selects the path to propagate the information towards the output. As for the case of the Accumulator, the figure shows only one unit, while in the complete CapsAcc architecture there is one Activation Unit per each column of the systolic array. The 25-bits data values coming from the Accumulators are reduced to an 8-bit fixed-point value, to reduce the computations.

Note: the Rectified Linear Unit (ReLU) [10] implements a simple transformation function which maps all the negative values to zero while not affecting the positive ones. It is implemented simply using a one bit comparator which makes the decision based on the sign bit. This function is used for every feature of the first two layers of the CapsuleNet.

We designed the **Normalization operator (Norm)** with a structure similar to the Multiply-and-Accumulate operator, where, instead of a traditional multiplier, there is the *Power2* operator. Its data path is shown in Figure 9f. A register stores the partial sum and the *Sqrt* operator produces the output. We designed the square operator as a Look-Up Table with 12-bit input and 8-bit output. It produces a valid output every $n+1$ clock cycles, where n is the size of the array for which we want to compute the Norm. This operator is used either as it is to compute the classification prediction, or as an input for the Squashing function.

We designed and implemented the **Squashing function** as a Look-Up Table, as shown in Figure 9e. Looking at Equation (1), the function takes an input s_j and its norm $||s_j||$. The Norm input is coming from its respective unit. Hence, this Norm operation is not implemented again inside the Squash unit. The LUT takes as input a 6-bit fixed-point data and a 5-bit fixed-point norm to produce an 8-bit output. We decided to limit the bit-width to reduce the computational requirements at this stage, following the analysis performed in Section III that shows the highest computational load for this operation. A valid output is produced with just one additional clock cycle compared to the Norm.

The **Softmax function** design is shown in Figure 9g. First, it computes the exponential function (8-bit Look-Up Table) and accumulates the sum in a register, followed by division. Overall, having an array of n elements, this block computes the softmax function of the whole array in $2n$ clock cycles.

D. Control Unit

At each stage of the inference process, it generates different control signals for all the components of the accelerator architecture, according to the operations needed. It is essential for correct operation of the accelerator.

V. Results and Discussion

A. Experimental Setup

We implemented the complete design of our CapsAcc architecture in RTL (VHDL), and evaluated it for the MNIST dataset *(to stay consistent with the original CapsuleNet paper)*. We synthesized the complete architecture in a 32nm CMOS technology library using the ASIC design flow with the Synopsys Design Compiler. We did functional and timing validation through the gate-level simulations using ModelSim, and obtained the precise area, power and performance of our design. The complete synthesis flow is shown in Figure 10, where the orange and blue colored boxes represent the inputs and the output results of our experiments, respectively.

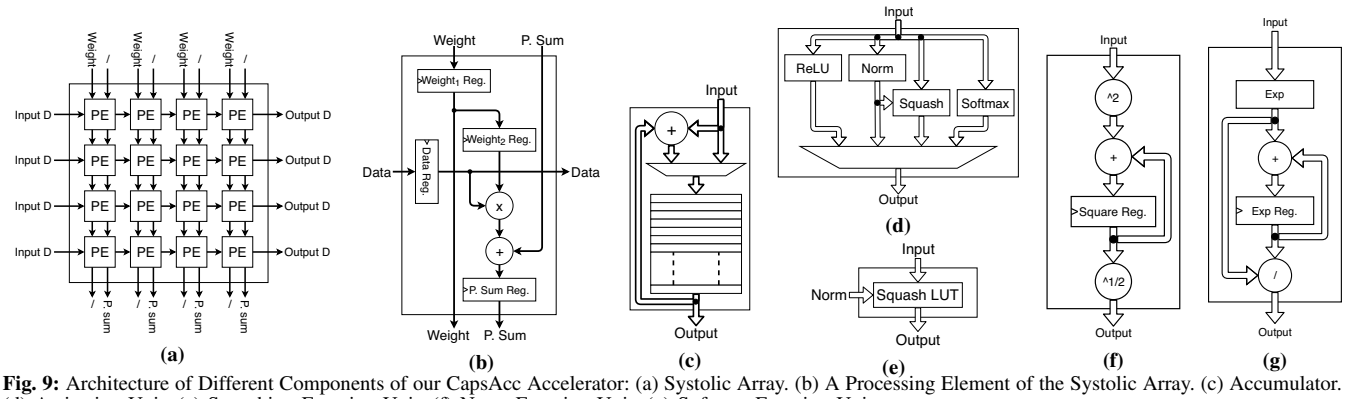

Fig. 9: Architecture of Different Components of our CapsAcc Accelerator: (a) Systolic Array. (b) A Processing Element of the Systolic Array. (c) Accumulator. (d) Activation Unit. (e) Squashing Function Unit. (f) Norm Function Unit. (g) Softmax Function Unit.

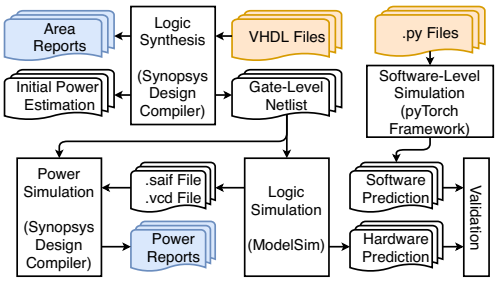

Fig. 10: Synthesis flow and tool chain of our experimental setup.

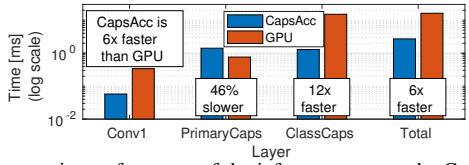

Fig. 11: Layer-wise performance of the inference pass on the CapsuleNet on our CapsAcc accelerator, compared to the GPU.

Important Note: since our hardware design is fully functionally compliant with the original CapsuleNet of the work of [12], we observed the same classification accuracy. Therefore, we do not present any classification results in this paper, and only focus on the performance, area and power results, which are more relevant for an optimized hardware architecture.

B. Discussion on Comparative Results

The graphs shown in Figure 11 report the performance (execution time) results of the different layers of CapsuleNet inference on our CapsAcc accelerator, while Figure 12 shows the performance of every sequence of the routing process. Compared with the GPU performance (see Figures 6 and 7), we obtained a significant speed-up for the overall computation time of a CapsuleNet inference pass (6×). *The main notable improvements are witnessed in the ClassCaps layer* (12×) *and in the Squashing operation* (172×).

C. Detailed Area and Power Breakdown

The details and synthesis parameters for our design are reported in Table II. Figure 13 shows the area and power breakdowns of our CapsAcc architecture. These figures show that the area and power contributions are dominated by the buffers, and the systolic array consumes just 1/4 of the total budget.

VI. Conclusions

We presented the first CMOS-based hardware accelerator for the complete CapsuleNet inference. To achieve high performance, our CapsAcc architecture employs a flexible systolic array with several optimized data-flow patterns that enable it to achieve higher parallelism for diverse operations of the CapsuleNet processing. Our results show a significant speedup compared to an optimized GPU implementation, thereby preserving the classification accuracy of the original CapsuleNet design of [12]. We also presented power and area breakdown of our hardware design. Our CapsAcc provides the first proof-of-concept for realizing CapsuleNet hardware, and opens new avenues for its high-performance inference deployments. An extended version is available in [9], for providing more technical details to the readers.

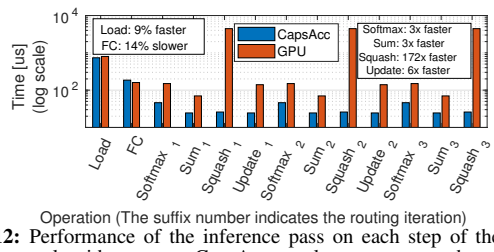

Fig. 12: Performance of the inference pass on each step of the routing-by-agreement algorithm on our CapsAcc accelerator, compared to the GPU.

Tech. node [nm]	32
Voltage [V]	1.05
Area [mm^2]	2.90
Power [mW]	202
Clk Freq. [MHz]	250
Bit width	8
On-Chip Mem. [MB]	8

TABLE II: Parameters of our synthesized CapsAcc accelerator

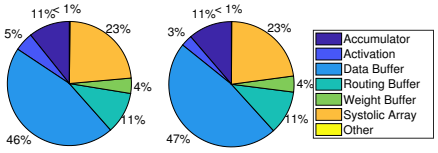

Fig. 13: (a) Area and (b) Power Breakdown of our CapsAcc Accelerator.

References

[1] Y. H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy efficient dataflow for convolutional neural networks. In *ISCA*, 2016.

[2] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. In *Neural Networks*, 2005.

[3] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efcient Inference Engine on Compressed Deep Neural Network. In *ISCA*, 2016.

[4] G. E. Hinton, A. Krizhevsky, and S. D. Wang. Transforming auto-encoders. In *ICANN*, 2011.

[5] N. P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[6] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.

[8] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *HPCA*, 2017.

[9] A. Marchisio, M. A. Hanif, and M. Shafique. CapsAcc: An Efficient Hardware Accelerator for CapsuleNets with Data Reuse. *arXiv preprint arXiv:1811.08932*, 2018.

[10] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[11] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *ISCA*, 2017.

[12] S. Sabour, N. Frosst, and G. E. Hinton. Dynamic routing between capsules. In *NIPS*, 2017.

[13] pyTorch framework: https://github.com/pytorch/pytorch