https://doi.org/10.2352/EI.2024.36.10.IPAS-255

Scalable and Efficient orchestration of machine learning workloads on DSPs with multi-level memory architecture

Aaron Sequeira, Febin Sam, Anshu Jain, Pramod Swami

Abstract

Deep learning has enabled rapid advancements in the field of image processing. Learning based approaches have achieved stunning success over their traditional signal processing-based counterparts for a variety of applications such as object detection, semantic segmentation etc. This has resulted in the parallel development of hardware architectures capable of optimizing the inferencing of deep learning algorithms in real time. Embedded devices tend to have hard constraints on internal memory space and must rely on larger (but relatively very slow) DDR memory to store vast data generated while processing the deep learning algorithms. Thus, associated systems have to be evolved to make use of the optimized hardware balancing compute times with data operations. We propose such a generalized framework that can, given a set of compute elements and memory arrangement, devise an efficient method for processing of multidimensional data to optimize inference time of deep learning algorithms for vision applications.

Introduction

Deep learning algorithms have been developed for applications such as object detection, segmentation etc. In the automotive industry particularly, there has been a push to deploy these algorithms for advanced driver assistance systems (ADAS). The requirement is to run deep learning algorithms on chip while minimizing energy consumption and maintain cost effectiveness. In practice, deep learning algorithms for computer vision are deployed on embedded devices for a variety of use cases such as:

1. Single network running on a single input stream.
2. Single network running on multiple input stream.
3. Multiple input running on multiple input streams.

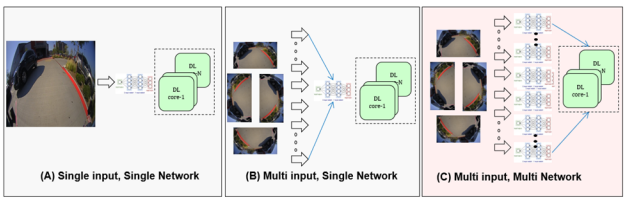

Figure 1. *Different use cases of deep learning models on embedded devices*

In order to effectively utilize the underlying hardware, careful examination of the entire use case is required to the derive optimal mapping and scheduling while efficiently utilizing available memory with a goal of providing most optimal solution. Since deep learning architectures process large, multidimensional data, it becomes imperative to create dataflows that can effectively use available memory and reduce latency due to I/O operations.

As such, the addition of new layer types to deep learning network architectures creates a constant need of developing new dataflows to make efficient use of available memory. Hence, there was a need to create a generic framework to represent the dataflow to enable quicker prototyping of new dataflows while ensuring low latency of execution due to the framework itself.

Related Work

Deploying learning-based vision models on embedded platforms has been an ongoing enterprise since the start of the deep learning age. Over time various architectures and associated software systems have been developed to accelerate the execution of deep learning algorithms. All implementations leverage the repetitive nature of computation in the algorithms, which typically are linear algebraic operations such as 2D convolution, pooling etc. The more popular implementations like CUDA [5] and OpenGL [1] utilize underlying GPUs to parallelize execution of learning algorithms across GPU cores. Whereas GPU based accelerators provide lower latencies, it comes at a cost of high energy consumption. Other implementations like AVX [2] utilize processor intrinsic SIMD vector instructions to enable some level of parallel execution. This somewhat mitigates the energy consumption issue but the latency of execution is much higher. There also exist a class of heterogeneous processors comprising specialized DSPs and CPUs that absorb the advantages of both of the earlier mentioned approaches. Such systems have become widespread in the embedded world due to the favorable trade-off between energy and latency of execution. The Texas Instruments' Jacinto and Sitara processors have implemented such heterogeneous processors as subsystems within their SOCs. However, most software implementations focus on reducing actual processor computation. To the best of the authors' knowledge, there has been no concerted focus on developing systems to coordinate data movement across memory regions with actual processor execution. We propose and develop such a mechanism in the following sections. Prior art referenced within this paper refers to adhoc decision making within the firmware to move and process data.

Proposed Representation

To get around the limitation of having to process the full input tensor to a layer, we generalize the representation of data using "workloads". We define a workload to be a container encapsulating the processing of a set of inputs generating exactly one output. There may be multiple inputs and each input may represent the full or part of a tensor. Using this workload representation, we can abstract the layer type as a parameter of the workload and keep the input tensor variable. A workload contains the following

constituent elements:

1. *Buffer*: A 2D object that defines the memory region containing a full or part tensor. The tensor within the buffer itself may be N-dimensional.
2. *Joint* : A structure containing access information of a tensor within a buffer. The joint is describes M-dimensions to access the tensor within the buffer as well as the offset required to jump between dimensions. It is not necessary that this maps exactly to the tensor. It is sometimes convenient to assume a flattened representation of the tensor for purposes of processing the data. For our implementation, we found it practical to restrict the dimensionality to 4.
3. *Link* : A structure that connects multiple source joints to a single sink joint. It is primarily responsible for defining timing information to execute the movement of data between the constituent buffers and relative to the other links present in the workload. The link also defines operations to be performed on source joints to generate the output data. Based on the kind of operation performed on the data, we further classify links into the following categories:
    - *Transfer Link*: Used to transfer data from source to sink and perform basic data operation. Generally implemented using DMA units.
    - *Processing Link*: Used to transform a series of input data. The exact operation is detailed as part of flags within the link. Implementation of processing link is by any specific processing unit.

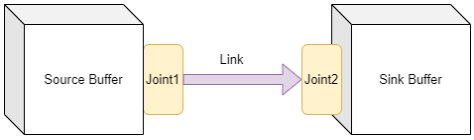

Figure 2. *Illustration of relation between links, joints and buffers*

Due to the nature of tensor processing in vision algorithms on hardware, we tend to process a larger amount of data in the first and last iterations as compared to the intermediate iterations. To incorporate this phenomenon, we allow joints to have 3 stages: pipeup, pipeline and pipedown. Thus each joint contained within a link can be viewed as an independent state machine moving through each of the three aforementioned stages.

Illustrative Example of Workload

Consider the processing of convolution layer. The size of the input tensor and weights is usually very large and may not fit on-chip. More often than not these tensors are stored in the DDR. To balance the access time against the size of memory on-chip, we bring part of the tensor to on-chip memory and process part-wise. Post processing, output may be reassembled part-by-part in the DDR. To express this process, we make use of the workload as depicted in Fig.3.

- Transfer links (**T1, T2, T3**) progressively bring in partial data from large DDR memory to much faster on chip memory (L2/L3).

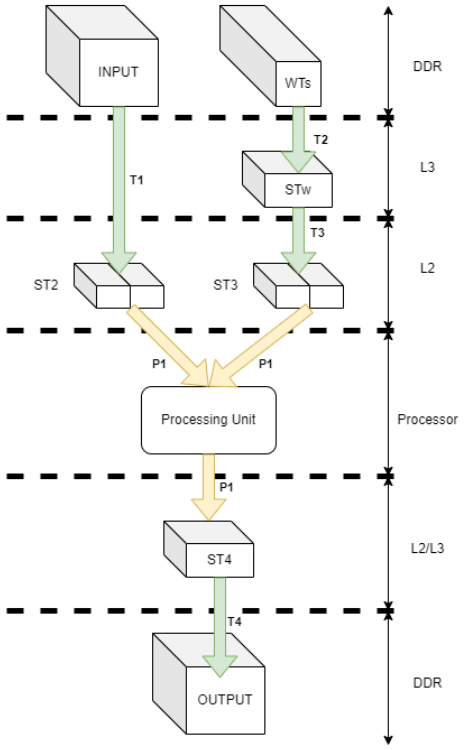

Figure 3. *Minimal workload representation showing buffers with associated memory spaces and connecting links*

Algorithm 1 UpdateJoint: Function to update state of joint

Require: $repeatIter < repeat$
  $iterCount \leftarrow iterCount + 1$
  if $iterCount > period$ **then**
    $freqIter \leftarrow freqIter + 1$
    $iterCount = 0$
  end if
  if $freqIter > freq$ **then**
    $repeatIter \leftarrow repeatIter + 1$
    $freq \leftarrow 0$
  end if
  if $freq = 0$ **then** ▷ Following updates the flow stage
    $stage \leftarrow$ PIPEUP
  else if $freq = freq - 1$ **then**
    $stage \leftarrow$ PIPEDOWN
  else
    $stage \leftarrow$ PIPELINE
  end if

Algorithm 2 ExecuteLink: Function to execute link

Require: $iterCount > delay$
  $jointIdx \leftarrow 0$
  while $jointIdx < numJoints$ **do**
    $updateJoint(joint[jointIdx])$
    $jointIdx \leftarrow jointIdx + 1$
  end while
  $link.funcExec(link)$ ▷ Executes link specific function

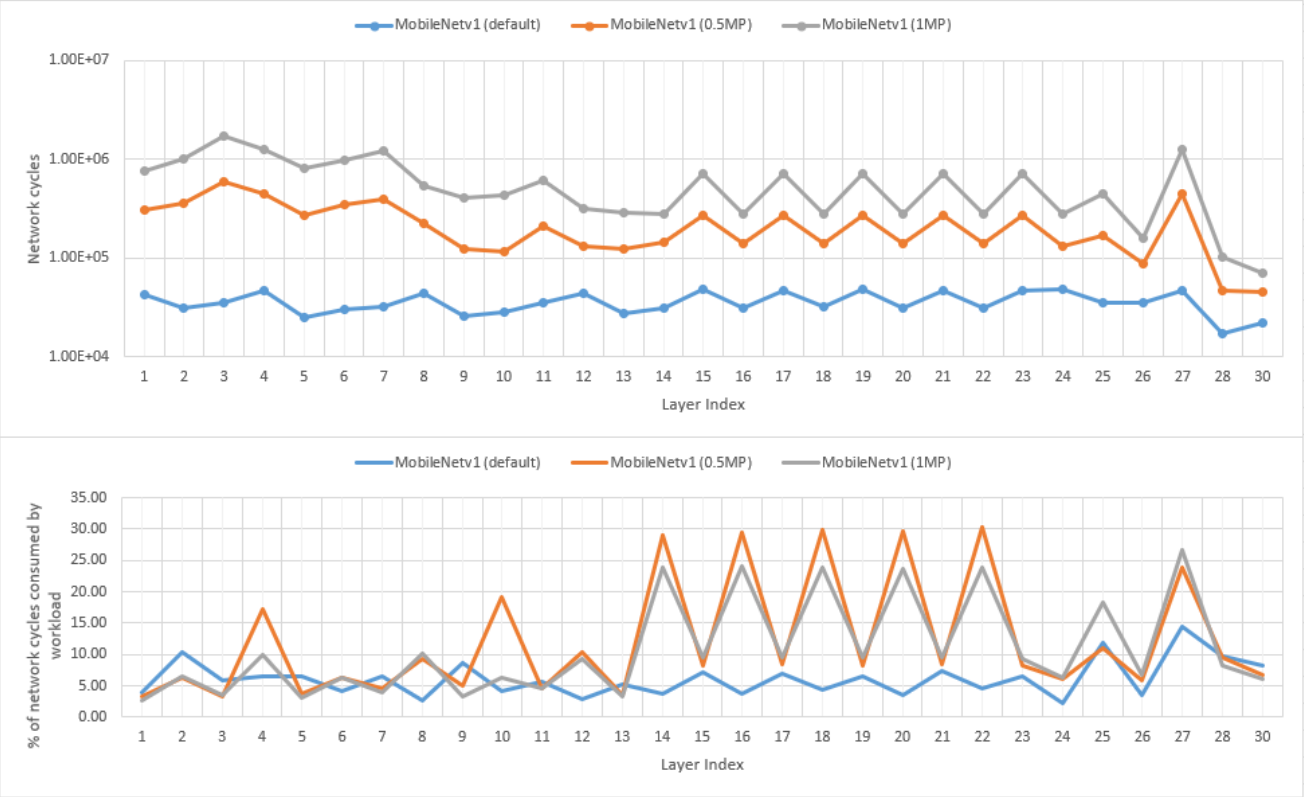

Figure 4. ***Top****: Network cycles consumed for each layer.* ***Bottom****: Percentage of cycles consumed by workload as a fraction of layer cycles.*

- Processing link (**P1**) depicts a generic processing unit (can be DSP, CPU, GPU etc) that performs some operation, in this case convolution, on input data and dumps the output into a partial output buffer in L3.
- Transfer link (**T4**) takes the output data generated by P1 and progressively reassembles it in DDR.

Timing Diagram

Following Algorithms 1 and 2, we define the time of activation of each link using the following table. By coordinating the execution between links using the delay, frequency and period variables (Algorithm 1), we complete processing of a block of data. An example of link activation can be seen if Fig.5 with reference to illustrative example of convolution workload in Fig.3.

I	0	1	2	3	4	...	n-4	n-3	n-2	n-1	n
T1											
T2											
T3											
P1											
T4											

Figure 5. *Timing Diagram of various links within the workload. Highlighted cells indicates period of activation.*

Observations

We present the performance of our proposed representation by running benchmark networks for deep learning on proprietary Texas Instruments SOCs. Specifically, data reported in Fig. 6 is taken from the TDA4VE device. To further illustrate the advantages of our proposed representation, we display the layer level performance of mobileNetv1 [3], a standard benchmark network for embedded vision in various input configurations.

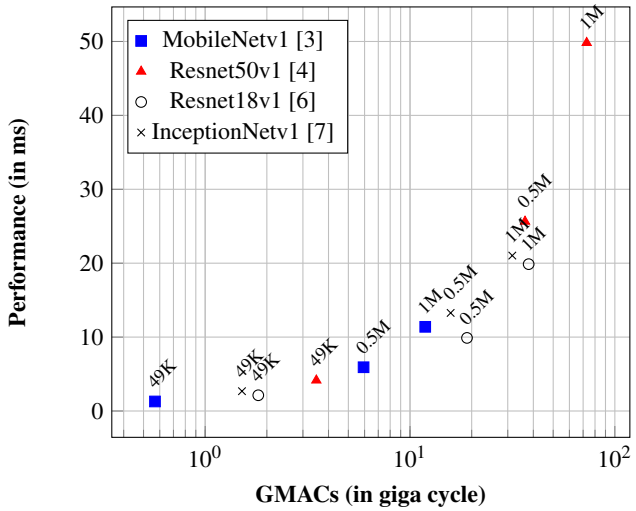

Figure 6. *GMACs vs time taken for execution of network on device for benchmark networks of labeled input plane sizes.*

From the graph in Fig. 6, we can clearly infer that for increasing input network input size, the execution of network on

chip scales proportionally. Specifically, we look at the default input configuration and network when input is scaled to 0.5MP and 1MP. We attach the layer level dimensions in Table 1.

Table 1: MobileNet Body Architecture [3]

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Note that, for the point of analysis we exclude fully connected layer due to workload integration for the specific dataflow being incomplete at the time of publication. As can be seen in Fig. 4, we can observe that there is an initial spike for workload cycles for higher resolutions as compared to the default dimensions. However, as the input size increases, the consumption of workload cycles as a percentage of overall layer level cycles falls.

Table 2: Comparison of code footprint size of firmware (in Kilobytes) of proposed implementation against prior art

Device	Prior Art	Workload	% change
TDA4VH	210.12	87.75	58.23
TDA4VM	116.50	57.3	50.81
TDA4VE	210.12	87.75	58.23
AM62A	211.31	85.94	59.33

Additionally, there is a significant fall in code memory footprint as can be observed from Table 2. The size decrease is attributed to the generalization of the dataflow mechanism and movement of decision-making code off the device.

Results

The proposed orchestration mechanism provides a simplistic way to create and execute specific dataflows while consuming less than 35% of overall network cycles when executed on TI SOCs. The proportion of network cycles consumed by the proposed workload falls with rising input sizes. The mechanism also allows for core dataflow related decisions to be moved off chip which results in over 50% reduction in code size across devices under consideration. Apart from the quantitative improvement, we find that it is easier to add newer dataflows without touching the firmware. Triaging dataflow failures has also become easier since the dataflow is decided off-chip as compared to earlier implementation.

The proposed mechanism has been implemented on Texas Instruments SOCs: TDA4VH, TDA4VE, TDA4VM and AM62A devices.

Acknowledgments

We would like to thank Mr. Shyam Jagannathan, who is a Senior Member of Technical Staff in the Embedded Processors Group at Texas Instruments, for presenting our work at the Electronic Imaging conference, 2024.

References

[1] Kurt Akeley. *The OpenGL graphics system: a specification*. June 1992. URL: https://www.microsoft.com/en-us/research/publication/the-opengl-graphics-system-a-specification/.

[2] Takuya Edamatsu and Daisuke Takahashi. "Fast Multiple-Precision Integer Division Using Intel AVX-512". In: *IEEE Transactions on Emerging Topics in Computing* 11.1 (2023), pp. 224–236. DOI: 10.1109/TETC.2022.3196147.

[3] Andrew Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: (Apr. 2017).

[4] Miguel Lopez-Montiel et al. "Evaluation Method of Deep Learning-Based Embedded Systems for Traffic Sign Detection". In: *IEEE Access* 9 (July 2021), pp. 101217–101238. DOI: 10.1109/ACCESS.2021.3097969.

[5] John Nickolls et al. "Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?" In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. URL: https://doi.org/10.1145/1365490.1365500.

[6] Allena Venkata Sai Abhishek. "Resnet18 Model With Sequential Layer For Computing Accuracy On Image Classification Dataset". In: 10 (July 2022), pp. 2320–2882.

[7] C. Szegedy et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2015.7298594.

Author Biography

Aaron Sequeira received his B.Tech degree in Electronics and Communication Engineering from the National Institute of Technology Karnataka (2021). He currently works at Texas Instruments as an embedded software engineer working on optimizing on-chip dataflows for computer vision tasks. His areas of interest are deep learning, computer

vision and embedded systems.

Febin John Sam received his B.Tech in Electrical and Electronics Engineering from National Institute of Technology Dimapur (2019) and his M.Tech in Computer Science Engineering from Indian Institute of Technology Kharagpur (2021). He now works at Texas Instruments in the Embedded Processing Analytics Technology Division. His work has focused on Memory Management on DSPs, to enable low latency inference of Deep Learning Computer Vision networks.

Anshu received his B.Tech and M.Tech in electrical engineering from Indian Institute of technology Bombay (2008). He is currently a senior embedded engineer at Texas Instruments India and has over 12 years of experience in the field. His domains of interest are signal processing specifically around computer vision, radar and deep learning.

Pramod Swami is a Distinguished Member of Technical Staff at Texas Instruments (TI), leading the software development for EdgeAI processing. His domains of interest are Embedded systems, Digital Signal Processors, Deep Learning, Computer Vision, Image Processing, and Video coding. He received his Bachelor's degree in Electronics and Communication Engineering from Malaviya National Institute of Technology (MNIT) Jaipur in 2001. He holds close to 50 USPTO patents and 25+ prestigious conference papers.