Work-in-Progress: DORY: Lightweight Memory Hierarchy Management for Deep NN Inference on IoT Endnodes

Work-in-Progress: DORY: Lightweight Memory Hierarchy Management for Deep NN Inference on IoT Endnodes

Alessio Burrello
alessio.burrello@unibo.it
DEI, University of Bologna
Bologna, Italy

Francesco Conti
fconti@iis.ee.ethz.ch
DEI, University of Bologna
Bologna, Italy

Angelo Garofalo
angelo.garofalo@unibo.it
DEI, University of Bologna
Bologna, Italy

Davide Rossi
davide.rossi@unibo.it
DEI, University of Bologna
Bologna, Italy

Luca Benini
lbenini@iis.ee.ethz.ch
DEI, University of Bologna
Bologna, Italy
IIS, ETH Zurich
Zurich, Switzerland

ABSTRACT

IoT endnodes often couple a small and fast L1 scratchpad memory with higher-capacity but lower bandwidth and speed L2 background memory. The absence of a coherent hardware cache hierarchy saves energy but comes at the cost of labor-intensive explicit memory management, complicating the deployment of algorithms with large data memory footprint, such as Deep Neural Network (DNN) inference. In this work, we present *DORY*, a lightweight software-cache dedicated to DNN Deployment Oriented to memoRY. DORY leverages static data tiling and DMA-based double buffering to hide the complexity of manual L1-L2 memory traffic management. DORY enables storage of activations and weights in L2 with less than 4% performance overhead with respect to direct execution in L1. We show that a 142 kB DNN achieving 79.9% on CIFAR-10 runs 3.2× faster compared to its execution directly from L2 memory while consuming 1.9× less energy.

1 INTRODUCTION

[1] In the IoT scenario, Deep Learning (DL) algorithms are considered attractive for edge processing, thanks to their capability to filter/compress raw data into a much more semantically dense and rich format. However, *i*) the small on-board memory, *ii*) the limited computational capabilities, and *iii*) the battery constraints typical of MCU-class devices deployed as end-nodes demand aggressive software and algorithmic optimization.

To achieve high performance and energy efficiency and tackle the challenges above, recently proposed IoT end node architectures leverage parallel computing by multiple cores on a small, ultra-fast L1 scratchpad memory – while most data resides on a larger, slower L2. However, this solution significantly complicates programming as these architectures give up energy-expensive coherent data caches. To solve this problem, SW-Caches have been proposed on top of these architectures [6, 7]. Specifically, to ease the deployment of high memory footprint DL based algorithms, ad-hoc memory management flows have been developed to minimize inference time [2, 4], by exploiting their regularity through the data reuse and an optimal CNN scheduling [5].

[1] This work was partially supported by the EU H2020-ECSEL project AI4DI (g.a. 826060) and by the Open transPREcision COMPuting (OPRECOMP) Project funded by the European Union's Horizon 2020 Research and Innovation Program (g.a. 732631).

CODES/ISSS '19, October 13–18, 2019, New York, NY, USA

ACM ISBN 978-1-4503-6923-7/19/10…$15.00
https://doi.org/10.1145/3349567.3351726

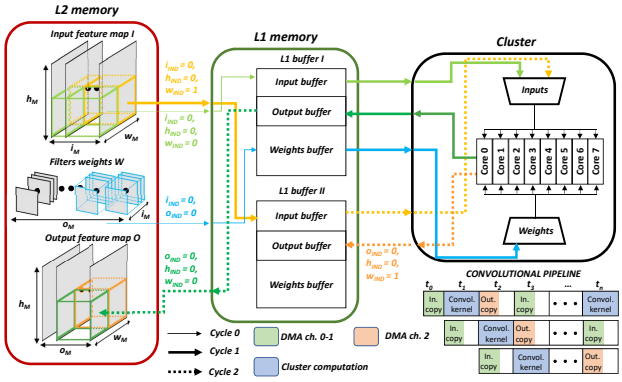

Figure 1: DORY flow. Left: the DMA manages L2-L1 communication using double-buffering. Right: the cores compute a kernel on the current tile stored in one of the L1 buffers.

In this work, we present work-in-progress results on a novel lightweight SW-cache for scratchpad memory management dedicated explicitly to DNN *Deployment Oriented to memoRY*. DORY, leveraging static data tiling and DMA-based prefetching and double buffering techniques, completely hides the data memory transfer overhead from L2 to L1 memory and vice-versa. Moreover, DORY exploits a constraint optimizer to vertically integrate memory limitations with DNN kernels requirements by finding an optimal tiling strategy to achieve the highest performance in MAC/cycle. Relying on high-performance software backend for parallel execution of DNN kernels, DORY enables the inference of DNNs that cannot fit within the L1 memory, outperforming the execution from L2 memory by 3.2× in terms of inference latency while saving 1.9× energy.

2 MEMORY MANAGEMENT FRAMEWORK

In the following, we provide an overview of the two main blocks of DORY.

a) DORY Optimizer. It takes as input the maximum dimension of the L1 buffer and the layer constraints from the NN topology (a PyTorch layer), such as the number of channels, padding, and spatial dimensions. Moreover, we include in the optimizer the requirements of the software back-end kernel, by inserting different optimization criteria to maximize the back-end performance/energy-efficiency. We abstracted this problem as an integer constraint problem; in the optimizer, we use the open-source OR-tools from Google AI [1] to solve it, matching constraints (i.e. maximum tile dimensions) and objectives (i.e. performance). We took advantage of this fact to tailor the optimizer to generate tiles that are better suited for the computation strategy (output-stationary with HWC layout) used

```
LTO: for (o=0; o<o_M; o++)              // output chan tiles
     LTH: for (h=0; h<h_M; h++)         // spatial height tiles
     LTW: for (w=0; w<w_M; w++)         // spatial width tiles
          LTI: for (i=0; i<i_M; i++)    // input chan tiles
               dma_wait (X_L1_load); swap (X_L1_load, X_L1_exec)
               dma_async (X_L1_load ← X_L2[i,w,h])
               dma_wait (X_L1_load); swap (W_L1_load, W_L1_exec)
               dma_async (W_L2[i,o] ← W_L1_load)
               if (o + h + w + i > 0)
                    DNN_kernel (X_L1_exec, W_L1_exec, Y_L1_exec)
          // from 3° cycle - fully operating pipeline
          if (o + h + w + i > 1)
               dma_wait (Y_L1_load); dma_async(Y_L1_load → Y_L2[o,w,h])
          swap (Y_L1_load ,Y_L1_exec)
```

Figure 2: DORY loop nest implementing the double buffering scheme.

Table 1: Energy and inference time for a 142 kB CNN.

	inf. time	energy	MAC/cycle/core
L2 fetching	22.7 ms	0.97 mJ	0.42 MAC
DORY	7.1 ms	0.51 mJ	1.37 MAC

in the computational kernels, maximizing their performance and not only fitting the L1.

b) DORY SW-Cache. To minimize inference overhead when embedding a DNN layer that doesn't fit the dimension of the L1 memory, DORY automatically generates code tailored to execute with minimal overhead the target layer, automatically instantiating asynchronous DMA transfers and double buffering as well as the calls to the underlying layer kernels, without any additional effort of the programmer. Fig. 2 provides DORY's scheduling scheme through LTO, LTW, LTH, and LTI loops on output channels, height, width, input channels tiles respectively. Loop iteration limits are automatically computed (as a consequence of the tile dimensions) by the DORY optimizer to minimize the overhead by taking into account both the features of the DNN layer (padding, stride, . . .) and the performance of the back-end kernel. Moreover, DORY autonomously controls the complete execution of the layer, by managing padding, stride, and overlap for each single tile (e.g. padding > 0 for border tiles whereas padding = 0 for internal ones, when the input padding parameter is > 0) and by managing double buffering in the the computation to completely hide the L2-L1 memory transfers.

Notice that the addition of the pipelined double-buffered memory transfers (Fig. 1) causes only minimal imbalance, as all DMA transfers are asynchronous and non-blocking. Fig. 1 also describes the memory transfers and the computation made in the first three cycles of the flow. Considering a full operatively pipeline, i.e. from the third cycle, we have three different parallel steps: (1) a new computation starts and fill the output buffer that was not used in the previous cycle; (2) the results of the last cycle are stored in L2 via DMA; (3) a new set of inputs is loaded in L1 via one other DMA channel. At each cycle, we swap the load and the execution buffer (*swap* operation of Fig. 2) to enable double buffering. The process continues until the LTO loop ends, i.e., when all the outputs reside in L2 memory.

3 RESULTS & CONCLUSIONS

We evaluate the performance of DORY on GreenWaves Technologies GAP-8 [3] processor, which is based on the eight-cores PULP architecture and features 64 kB of L1 (single-cycle latency) and 512 kB of L2 memory (fifteen-cycle latency), using highly optimized software back-end for DNN kernels. All the kernels are quantized to 8-bit, in both activations and weights, and reach up to 1.8 MAC/cycle/core (the baseline performance is 1 MAC/cycle/core: 2 load, 1 SIMD-MAC and 1 store to compute four 1-byte MACs).

Fig. 3 depicts the performance of DNN layers when different resources are exploited: L1 fetching (assuming that all data resides in L1 memory and there are not load-use stalls), L2 fetching (assuming

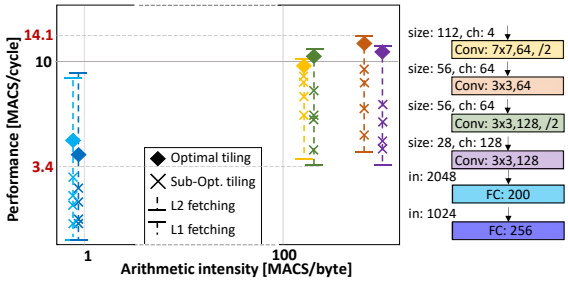

Figure 3: Performance evaluation of quantized 8-bits convolutional/linear layers.

all data resides in L2), and DORY data flow with or without DORY optimizer. For *CONV* layers, less than 4% performance degradation has been observed due to our added SW-cache. Most performance losses with respect to the roof of ~14 MACs/cycle are due to one of several effects. First, when the number of channels is very low, the *CONV* kernels cannot exploit parallelism, and efficiency partially drops (e.g. the first layer in Fig. 3). Second, border tiles result in very small input sets, therefore lowering the operational intensity and reducing performance (e.g. the third layer in Fig. 3). On the other hand, the memory-hungry linear layers demonstrate a higher overhead (40%-60%). This effect is due to the low operational intensity characterizing *FC* kernels (1 MAC/byte vs. 300 MAC/byte for convolutions), which does not allow to entirely hide the L2-L1 memory transfers and to the overhead of the DORY scheduling; in terms of absolute number of cycles, DORY requires less than 1.5K cycles for each loop over a tile, which is, however, not negligible for *FC* layers.

Fig. 3 also shows that the DORY optimizer finds the optimal solution (♦) in terms of MAC/cycle performance for different layers, which is not always trivial. Conversely, crosses (×) highlights that the effect of a sub-optimal tiling can be very detrimental, losing up to 3.5× MAC/cycle compared to the optimal solution.

Table 1 depicts the performance of DORY when running a full 142 kB DNN not fitting L1 memory on the GAP-8 IoT processor at 170 MHz and 1 Volt. Thanks to DORY, the inference of an image takes as little as 7.1 ms and consumes 0.51 mJ, outperforming the corresponding L2 fetching, by running 3.2× faster, while saving 1.9× energy. Moreover, DORY automatically generates the code, by reducing the burden of the programmer to write the kernel and the memory management single operations: from a PyTorch layer and the L1 memory dimension, DORY provides the function interface to be included in the code to run the layer.

These initial results showcase how the DORY double-buffering based SW-Cache can effectively hide the L2-L1 communication, achieving lower than 4% overhead in terms of time compared to the "ideal" pure L1 execution. Our current work focus in particular on extending automatic SW-based DNN caching on networks that do not fit even in L2 by enabling non-blocking two-level buffered data fetching from an external L3 DRAM.

REFERENCES

[1] Google AI. 2015. OR tools. (2015). https://developers.google.com/optimization/
[2] Leonardo Cecconi et al., 2017. Optimal tiling strategy for memory bandwidth reduction for cnns. In *International Conference on ACIVS*. Springer, 89–100.
[3] E. Flamand et al., 2018. GAP-8: A RISC-V SoC for AI at the Edge of the IoT. In *2018 IEEE 29th ASAP International Conference*. 1–4.
[4] Daniele Palossi et al., 2019. A 64mW DNN-based Visual Navigation Engine for Autonomous Nano-Drones. *IEEE Internet of Things Journal* (2019).
[5] Maurice Peemen et al., 2013. Memory-centric accelerator design for convolutional neural networks. In *2013 IEEE 31st ICCD*.
[6] Christian Pinto et al., 2013. A highly efficient, thread-safe software cache implementation for tightly-coupled multicore clusters. In *IEEE 24th ASAP Conference*
[7] Selma Saidi et al., 2013. Optimizing two-dimensional DMA transfers for scratchpad Based MPSoCs platforms. *Microprocessors and Microsystems*.