Performance evaluation of acceleration of convolutional layers on OpenEdgeCGRA

Performance evaluation of acceleration of convolutional layers on OpenEdgeCGRA

Nicolò Carpentieri[1], Juan Sapriza[2], Davide Schiavone[2], Daniele Jahier Pagliari[1], David Atienza[2], Maurizio Martina[1], Alessio Burrello[1]
[1] Politecnico di Torino, Torino, Italy, [2]EPFL, Lausanne, Switzerland
juan.sapriza@epfl.ch

Abstract

Recently, efficiently deploying deep learning solutions on the edge has received increasing attention. New platforms are emerging to support the increasing demand for flexibility and high performance. In this work, we explore the efficient mapping of convolutional layers on an open-hardware, low-power Coarse-Grain Reconfigurable Array (CGRA), namely OpenEdgeCGRA. We explore both direct implementations of convolution and solutions that transform it into a matrix multiplication through an Im2col transformation, and experiment with various tensor parallelism axes. We show that for this hardware target, direct convolution, coupled with weight parallelism reaches the best latency and energy efficiency, outperforming a CPU implementation by 3.4× and 9.9× in terms of energy and latency, respectively.

1 Introduction

The shifting paradigm from cloud to edge computing led to executing deep learning models into low-power and memory-tight devices. Among the plethora of models, Convolutional Neural Networks (CNNs) have emerged as one of the most powerful tools for various tasks, including image recognition, and natural language and signal processing. However, their efficient execution on resource-constrained edge devices remains a significant challenge given the high memory footprint and the huge number of operations. To cope with this challenge, various architectures have been proposed. ASICs offer the highest performance and energy efficiency for specific applications due to their custom-tailored hardware design [1]. FPGAs, on the other hand, provide increased flexibility through their fine-grained reconfigurable architecture, making them more versatile than ASICs but generally less energy efficient [2]. In this work, we explore the use of Coarse-Grain Reconfigurable Arrays (CGRAs) as potential candidates to cover a different part of the design space of edge computing systems (i.e., considering trade-offs between performance, energy efficiency, area and versatility) to execute CNN applications [3]. CGRAs present programmable hardware that can be customized to specific tasks, making them well-suited for dynamic edge computing environments. Thus, they offer an attractive meet-in-the-middle solution between performance, efficiency, and task-versatility. Nevertheless, the efficient mapping of convolutional layers onto CGRAs is essential to exploit their potential benefits. Figure 1 shows an example of the mapping of a convolution on our target CGRA.

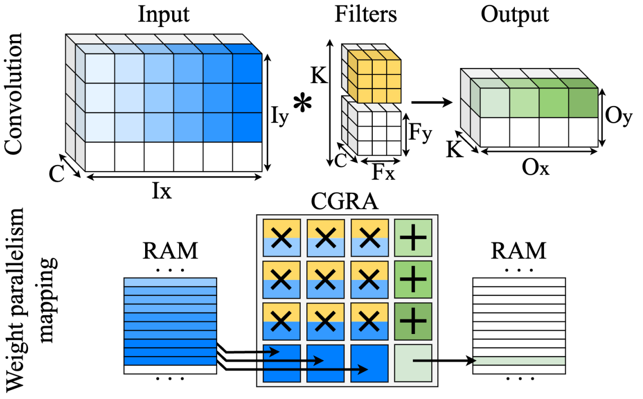

Figure 1: (Top) 2D convolution scheme. (Bottom) Direct convolution with weight parallelism. Nine PEs perform dot products between constant weights and sequentially loaded inputs. The other PEs load new inputs or sum partial outputs.

Although several works have focused on mapping techniques for high performance CGRAs [4] or specializing CGRAs for deep learning [3, 5], low-power (mW-power), general-purpose CGRAs have not been explored equally for tinyML [6]. To fill this gap, this work addresses the problem of mapping convolutions, focusing on the OpenEdgeCGRA architecture for edge computing applications [7]. Our goal is to outline efficient practices that allow to leverage this accelerator, minimizing the impact of the overheads it imposes. For this reason, we investigate various state-of-the-art computational and memory management strategies, aiming to uncover the most efficient mapping technique that balances performance and resource constraints. Specifically, we present a two-fold contribution: (i) we explore different implementation paradigms for convolution and different tensor parallelism axes; (ii) we benchmark the results of the different implementations, measuring energy, latency, performance, and memory usage, and provide insights on the best mapping technique for low-power CGRA. This analysis highlights the predominance of direct convolution, coupled with weight parallelism, which reaches up to 3.4× and 9.9× in terms of energy and latency, respectively, compared to a plain CPU implementation, achieving an overall average performance of $0.6^{MAC}/_{cycle}$.

CF '24, May 7-9, 2024, Ischia, Italy

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

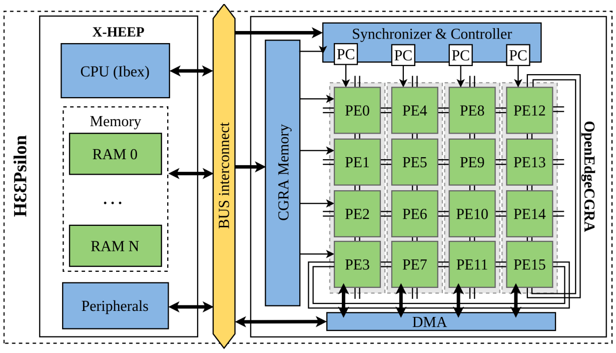

Figure 2: Architecture of the H$\mathcal{EE}$Psilon platform used as a test bench for this analysis, where the OpenEdgeCGRA is instantiated along with X-HEEP.

2 Material and Methods

In this section, we first introduce our target CGRA. Then, we describe the different convolutional kernels mapping techniques explored to exploit its hardware resources maximally.

2.1 Hardware Platform

In this work we target the OpenEdgeCGRA architecture: a low-power, general purpose, scalable, instruction-based CGRA. Although designed to execute health applications in low area and power footprint, thus not ideal for running regular high intensity kernels such as CNNs, this CGRA has been chosen for being open source, validated in silicon, and already integrated into a RISC-V microcontroller (X-HEEP) in the *H$\mathcal{EE}$Psilon* platform.All our work is open-source[1].

Figure 2 illustrates this platform, which includes a CPU and memory allowing for real application evaluation [8]. The CPU configures the CGRA and loads instructions into the CGRA's memory before launching its first execution. The instruction set of the CGRA includes 32-bit integer arithmetic and logical operations and supports both conditional and unconditional jumps, allowing the implementation of deep learning kernels. However, a Multiply-and-Accumulate (MAC) instruction which could lead to higher performance is not supported. In this work, we used an instance of the OpenEdgeCGRA with a 4×4 matrix of Processing Elements (PEs). Each PEs is composed of one Arithmetic-Logic Unit (ALU), two multiplexed inputs, one output register, a four-element Register File (RF), and a 32-word private program memory. The PEs are connected to their neighbors through a torus interconnect, allowing for reuse of neighbor data. Although columns in OpenEdgeCGRA have their own independent Program Counter (PC), we always used the four columns as part of a single application, which means that the latency of execution of a single CGRA-instruction is determined by the latency of the slowest operation among the 16 PEs. Each column of the OpenEdgeCGRA has a port connected to a Direct Memory Access (DMA) block, allowing them to access the memory subsystem.

2.2 Convolution mapping strategies

Our work explores different optimization directions to map a convolutional kernel onto the OpenEdgeCGRA. In particular, we first explore the implementation paradigm, i.e., the data layout coupled with its access order. Then, we investigate the computation's parallelization over the OpenEdgeCGRA's PEs. For all our experiments, we always consider convolutions with groups = 1, and a filter of dimension $F_X \times F_Y = 3 \times 3$ [9].

Convolution implementation: Image-to-Column (Im2col) vs Direct Access

We consider two different implementations, i.e., the direct convolution and the so-called Im2col transformation. The direct convolution does not manipulate the input data. It directly fetches data from memory, leading to non-sequential load operations of the input image and, therefore, higher overhead in data addressing. To minimize this overhead a Channel-Height-Width (CHW) data layout is typically used [10]. On the other hand, the Im2col transformation is the most adopted implementation in CPU and GPU kernel libraries, such as PULP-NN [11], Mxnet [12], or Tensorflow [13]. It transforms multi-channel 2D Convolutions into a vector-matrix product by turning each input activations' patch (originally a 3D tensor) into a 1D vector of dimension input channels (C) × filter rows (F_X) × filter columns (F_Y), which is multiplied with the 2D weights matrix, of dimension C×F_X × F_Y× output channels (K); note that this transformation simplifies the memory accesses, which become sequential. On the other hand, it requires more memory to store the buffer of reordered inputs and additional instructions to create this buffer, which could be non-negligible. We argue that Im2col transformation can leverage the loads with automatic index increment and parallel DMA ports of our target architecture. In [10], the authors show that the Height-Width-Channel (HWC) data layout is the most advantageous for the creation of the Im2col reorder buffer. Hence, we select it for our implementation.

Parallelization Axis

The main computation in a CNN involves six nested loops, which can be swapped and parallelized without changing the final result [11]. These loops correspond to i) output channels (K), ii) input channels (C), iii) output rows (O_X), iv) output columns (O_Y), v) filter rows (F_X), and vi) filter columns (F_Y). We explore the parallelization of the C, K, or $F_{X/Y}$ loops. Note that we do not explore parallelization of $O_{X/Y}$ loops given that it would enable reuse of neither the weights nor the inputs.

Weight Parallelism (WP): This method leverages the parallelization of the filter loops, F_X and F_Y. In this setup, each weight element of a single input and output channel is assigned to a different PE. For a 3×3 filter, this means that nine weight elements are distributed across nine PEs. Once these weights are retrieved from memory, the system performs multiple MAC operations by updating the inputs for each PE. The partial outputs generated by the PEs then move through the spatial array of the CGRA. This procedure is illustrated in Figure 1. In addition to the nine PEs engaged in computations, the final row (comprising three PEs) is tasked with updating the address to load the new input triplet (3×1), while the other 3×2 inputs can be efficiently reused by shifting them from the first two rows of PEs when computing the next output pixel on the same output image row. During this stage, the last column of PEs aggregates the nine computed partial sums and, if necessary, adds them to a previous partial sum when processing input channels $c_i > 0$. The last PE is designated for storing the accumulated partial

[1] Available on github.com/esl-epfl/HEEPsilon/tree/convolution_exploration

sum in memory. This cycle is repeated for the entire input spatial position before a new set of weights is loaded to process the next input channel. The outputs are sequentially generated starting from O_X, O_Y, and, finally, K. Importantly, this mapping scheme, which benefits from a CHW input layout, would not benefit from using the Im2col transformation.

Input-Channel Parallelism (IP): This method involves performing MAC operations relative to various input channels in parallel. It utilizes the Im2col technique to enable sequential access to the input and filter data. Note that using direct convolution for this parallelism strategy would be suboptimal given that the latency to access the data from each single PE would strongly increase given their storage position. In this mapping strategy, for each iteration of the most external loops (K, O_X, O_Y), every PE handles a distinct set of input channels (C/16 per PE) for the same output channel and spatial position. In the end, the partial sums computed by each PE are aggregated, and the next element is computed.

Output-Channel Parallelism (OP): This mapping aims to produce results simultaneously for different output channels. Its rationale is to minimize the latency for reading and writing partial sums by keeping them in the RF of each PE [14]. A different output channel is then assigned to each different PE, which stores a different set of weights, and the same input elements are broadcast to all PEs, to produce 16 output channels at the same spatial location in parallel. For this implementation, both the Im2col approach and the direct convolution implementations are considered.

Figure 3 shows the distribution of the operations in the innermost loop of each mapping strategy, over the CGRA's PEs. Note that while WP differs significantly, the other three mappings are identical: in the first two instructions, 16 inputs and weights are loaded (corresponding to 16 input or output channels). Next, the `mul` and `sum` operations are executed by all PEs. Then, in the last 5 instructions, the input and weight addresses and the iteration counter are updated, followed by the loop's branch instruction. Most PEs execute a `nop` during the last three instructions because only one to two PEs are in charge of updating the iteration counter and branching for the whole CGRA. Because of this bottleneck, the innermost loop reaches an overall PE utilization of 69%. For all three mapping strategies, this loop is repeated $F_X \times F_Y \times O_X \times O_Y \times C \times K/16$ times.

Conversely, the WP mapping is composed of a main internal loop and a border internal loop. The main loop is composed of only 4 instructions that allow the execution of the nine multiplications, the sum reduction, the load of a new input triplet, and the final store. However, as mentioned above, once a new output row has to be processed, also the other 6 inputs (2×3) should change, necessitating 5 additional instructions (border loop) to load the additional data and update indexes, as shown in the graph. In this case, the main loop is executed $O_X \times O_Y \times C \times K$ times with an utilization of 78%, while the border one is executed only once per row, i.e., $O_Y \times C \times K$ times.

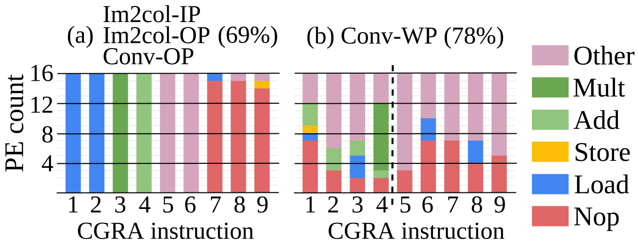

Figure 3: Operation distribution of different convolution mapping strategies. *Other* includes index updates, branch operations, and index manipulation.

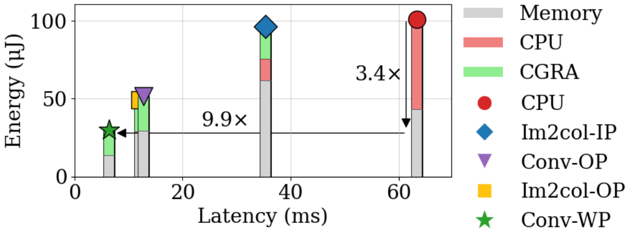

Figure 4: Energy vs. Latency comparison.

2.3 Evaluation Metrics

We compare our convolution mapping strategies in terms of the following metrics to provide a complete overview of the utilization and efficiency of the platform:

Latency: the time required to perform a complete convolution, comprising both the Im2col creation (if needed) and the kernel execution. The time required to load the instructions before the first iteration is neglected.

Energy: we consider the power consumption of a complete minimal system, including CGRA, CPU and memory subsystems. This allows for a fair comparison between different strategies but should not be used as a benchmark to compare the platform's efficiency, as this was not optimized. In the Im2col case, the MCU performs data reordering during the CGRA execution. At all other times the MCU enters a busy loop waiting for the CGRA interrupt.

Memory usage: to assess the scalability of each strategy, we characterize its memory footprint as the space required to store the input and output samples and the weight filters.

MAC/cycle: to compare the obtained execution speed with other state-of-the-art implementations, we compute the performance in terms of MAC operations per clock cycle ($MAC/_{cycle}$).

Behavioral simulation of each approach was performed using the OpenEdgeCGRA simulator[2]. Latency measurements were obtained from the FPGA implementation of *HEEPsilon* and validated against pre-synthesis simulation. The average power was obtained from post-synthesis simulation on a TSMC 65 nm technology process, where the CGRA required an area of $\sim$ 0.4 mm^2.

3 Experimental results

In this section, we present results concerning the latency and energy efficiency of the kernels described in Section 2, with a final analysis of their robustness to hyper-parameters variations. All kernels use 32-bit integer data.

3.1 Energy and latency evaluation

We run a baseline convolution with C=K=O_X=O_Y=16, and a 3×3 filter. For each mapping method, we measure execution latency and energy consumption of the three main blocks involved: CGRA, CPU,

[2] Available on github.com/esl-epfl/ESL-CGRA-simulator

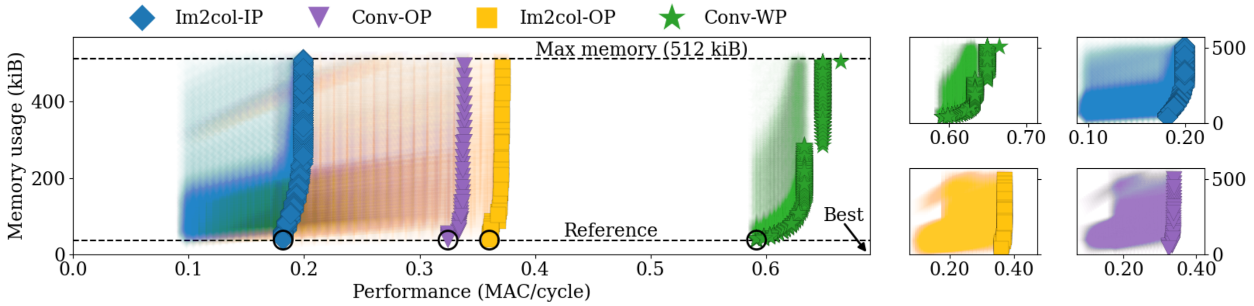

Figure 5: Impact on memory and performance of different hyperparameters. Pareto-optimal results are highlighted with a greater color intensity. The experiments of Section 3.1 are highlighted by black circles.

and memory. In Figure 4 results are compared against a CPU-only implementation. The WP approach reaches energy and latency improvements of 3.4× and 9.9×, respectively, at an average power of 2.5 mW, the highest among the CGRA-approaches. Its main advantage over the other strategies lies in the high data reuse rate of the weight stationary strategy, which brings two-fold benefits. First, the reduced number of memory accesses and their distribution over time avoids collisions between PEs, hence decreasing the latency of the whole computation. Secondly, the lower number of memory accesses also reduces the dynamic energy consumed by the memory subsystem. Figure 4 highlights that the latter is the largest energy-wise discriminative factor between methods -which would have been overlooked by an isolated analysis of the CGRA performance-. For example, the higher energy consumption of the Im2col-OP approach is not associated with the CPU involvement (which includes the implementation of Im2col) but with the more frequent load instructions. In this case, the CPU contribution is negligible and allows marginal improvements in both energy and latency with respect to the Conv-OP approach. In contrast, the Im2col-IP method requires frequent computation of the Im2col, leading to a higher CPU activity and doubling memory consumption. This situation also increases latency due to the overhead of launching each iteration. In this method, every call of the Im2col function creates one output position at a time and, additionally, each Im2col input organization has to be repeated for every output channel. Instead, Im2col-OP operates in parallel across output channels, allowing it to generate 16 output positions simultaneously with just one Im2col setup.

3.2 Robustness evaluation

We evaluate the performance deviation from the *baseline* case explored in the previous section by swiping the layer hyperparameters. We vary O_X and O_Y in [16, 64], C and K in [16, 144], increasing by 1 the dimension of each parameter until 32, and then in steps of 16 given the similar scalability. We limit our search to the maximum memory available in the system (512 kiB from $H\mathcal{E}\mathcal{E}Psilon$'s RAM banks). The results, illustrated in Figure 5, show that WP has the greatest robustness to hyperparameter changes, with increasing layer dimensions always leading to improved performance. WP remains the best approach for any hyperparameter combination, reaching up to $0.665 ^{MAC}/_{cycle}$ with C =16, K =16, and $O_X = O_Y$ = 64. It is noteworthy that increasing O_X and O_Y translates into an improvement in performance for the WP case thanks to two different contributions: first, the larger the input size, the higher the reuse of the loaded weights; second, a larger feature map reduces the occurrence of row changes while swiping the input activations, thus, the associated overhead of border loop (cf. Section 2.2).

On the other hand, all the other approaches see a drop in performance every time their parallelization dimensions are not a multiple of the number of PEs (i.e., 16), reaching their lowest performance ($\sim 0.1 ^{MAC}/_{cycle}$) when the parallelization dimension is equal to 17 due to the strong imbalance in the workload distribution. In this case, the Im2col-OP results the least robust with a performance reduction of 3.62× when compared to its best case.

4 Conclusions

This work has thoroughly analyzed the impact of different convolution mapping techniques, typical in CNN applications, on the OpenEdgeCGRA. Latency, energy, memory usage, and performance were evaluated to conclude that the WP approach is the best performing one, with a peak of $0.665 ^{MAC}/_{cycle}$. Energy and latency improvements compared to the CPU-only implementation reach 3.4× and 9.9×, respectively. While specialized architectures in the state of the art reach 23.3× higher performance [11], this work shows how to leverage the existing trade-offs between performance, smaller area (0.4 mm^2) and low-power (< 2.5 mW) on CGRAs. Thus, we underscore how such platforms are viable architectural options for heterogeneous edge AI accelerators to complement ultra-low-power microcontrollers.

References

[1] Norman P Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proc. of the 44th ISCA*, pages 1–12, 2017.

[2] Yixing Li et al. A gpu-outperforming fpga accelerator architecture for binary convolutional neural networks. *ACM JETC*, 14:1 – 16, 2017.

[3] Jungi Lee and Jongeun Lee. Specializing cgras for light-weight convolutional neural networks. *IEEE TCAD*, 41(10):3387–3399, 2021.

[4] Dhananjaya Wijerathne et al. Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(10):3290–3303, 2021.

[5] Christian Heidorn et al. Design space exploration for layer-parallel execution of convolutional neural networks on cgras. In *Proc. of the 23th SCOPES*, pages 26–31, 2020.

[6] Artur Podobas et al. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743, 2020.

[7] Rubén Rodríguez Álvarez et al. An open-hardware coarse-grained reconfigurable array for edge computing. In *Proc. of the 20th ACM CF*, pages 391–392, 2023.

[8] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. X-heep: An open-source, configurable and extendible risc-v microcontroller for the exploration of ultra-low-power edge accelerators. *arXiv preprint arXiv:2401.05548*, 2024.

[9] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*, volume 1. MIT Press, 2017.

[10] Liangzhen Lai et al. CMSIS-NN: efficient neural network kernels for arm cortex-m cpus. *CoRR*, abs/1801.06601, 2018.

[11] Angelo Garofalo et al. Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors. *Phil. Trans. of the Royal Society A*, 378(2164):20190155, 2020.

[12] Tianqi Chen et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[13] Sanjay Krishnan et al. Artificial intelligence in resource-constrained and shared environments. *SIGOPS Oper. Syst. Rev.*, 53(1):1–6, jul 2019.

[14] Vivienne Sze et al. *Designing DNN Accelerators*, chapter 5, pages 73–118. Springer Cham, 2022.