Parallelized Convolutions for Embedded Ultra Low Power Deep Learning SoC

Lorenzo Cunial, Ahmet Erdem, Cristina Silvano
DEIB – Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy
lorenzo.cunial@mail.polimi.it
ahmet.erdem@polimi.it
cristina.silvano@polimi.it

Mirko Falchetto, Andrea C. Ornstein, Emanuele Plebani, Giuseppe Desoli, Danilo Pau,
Advance System Technology, STMicroelectronics, Agrate Brianza, Italy
mirko.falchetto@st.com, andrea.ornstein@st.com, emanuele.plebani1@st.com, giuseppe.desoli@st.com, danilo.pau@st.com

***Abstract*—** Deep Convolutional Neural Networks (DCNNs) achieve state of the art results compared to classic machine learning in many applications that need recognition, identification and classification. An ever-increasing embedded deployment of DCNNs inference engines thus supporting the intelligence close to the sensor paradigm has been observed, overcoming limitations of cloud-based computing as bandwidth requirements, security, privacy, scalability, and responsiveness. However, increasing the robustness and accuracy of DCNNs comes at the price of increased computational cost. As result, implementing CNNs on embedded devices with real-time constraints is a challenge if the lowest power consumption shall be achieved. A solution to the challenge is to take advantage of the intra-device massive fine grain parallelism offered by these systems and benefit from the extensive concurrency exhibited by DCNN processing pipelines. The trick is to divide intensive tasks into smaller, weakly interacting batches subject to parallel processing. Referred to that, this paper has mainly two goals: 1) describe the implementation of a state-of-art technique to map DCNN most intensive tasks (dominated by multiply-and-accumulate ops) onto Orlando SoC, an ultra-low power heterogeneous multi cores developed by STMicroelectronics; 2) integrate the proposed implementation on a toolchain that allows deep learning developers to deploy DCNNs on low-power applications.

Keywords: Deep Learning, Neural Network, Parallelism, ultra-low power, SoC

I. INTRODUCTION

DCNN based pipelines are very accurate and widely applied to a large number of hard-to-solve problems in classification, detection, recognition, analysis and, recently, even synthetic signals generation in computer vision, speech and audio applications, robotic motion, navigation, financial data analysis, medical diagnostics, and many more. A large number of neural network topologies and operators is described in the state of the art. Many of them rely on tensor convolutions and on various set of additional operators deployed in a sequence of processing steps called layers.

In most cases, the inherent parallelism and associativity of these computational structures offer many opportunities for efficient and high-performance implementations. At the same time too many degrees of freedom it makes difficult to derive a comprehensive set of parallelization strategies.

Mapping a set of algorithms onto a multi- or many-core heterogeneous platform often relies on parallel programming modelling, which allows the description and control of concurrency, communication, and synchronization aspects of the parallel entities.

Nowadays, it is common to find approaches for mapping highly data parallel applications based on task-based models such as Intel's Threaded Building Blocks (TBB) and NVIDIA's CUDA; other examples include OpenMP, Cilk, and OpenCL. Each programming model trades off flexibility, expressiveness and ease of mapping/scheduling decisions differently. Under these models, the programmer can partition the application into small independent tasks. A task scheduler that handles dependencies and maximize throughput controls their execution. It has been noted that GPU and CPU parallel architectures show hard limits in computation vs energy efficiency when implementing DCNN applications; on the other side, a very energy-efficient architecture might only partially adopt similar parallel approaches being resource constrained when compared to non-embedded CPUs and GPUs. It could benefit from an ad-hoc approach exploiting a platform dependent parallelism extraction due to the severe constraints of energy consumption and memory footprint.

A state of art ultra-low power DCNN hardware is the Orlando SoC by STMicroelectronics, introduced at the beginning of 2017[1]. Figure 1.a shows it. It integrates custom designed DSPs and an instance of a reconfigurable dataflow custom HW accelerator fabric. The SoC designed in FD-SOI 28nm silicon technology with low power features and adaptive circuitry support a wide voltage range from 1.1V down to 0.575V. The chip also includes four banks of SRAM with dedicated 64bits bus ports each one of 1Mbyte. Furthermore, it is equipped with streaming DMAs in order to provide high-speed data transfer between memories and accelerators.

978-1-5386-6282-3/18/$31.00 ©2018 IEEE

As shown in Figure 1.b, each DSP cluster is provided with 2 32-bit RISC DSPs (6uW/Mhz@0.6V, running up to 1GHz fabricated in FD-SOI 28nm technology), 4-way 16KB I-Cache, 64KB Local RAM (composed by two 32KB banks) and a shared 64KB RAM 32bit. Since convolutional layers accounts for more than 90% of total operations in typical DCNN topologies, parallelization of their computation is of paramount importance in order to accelerate their execution.

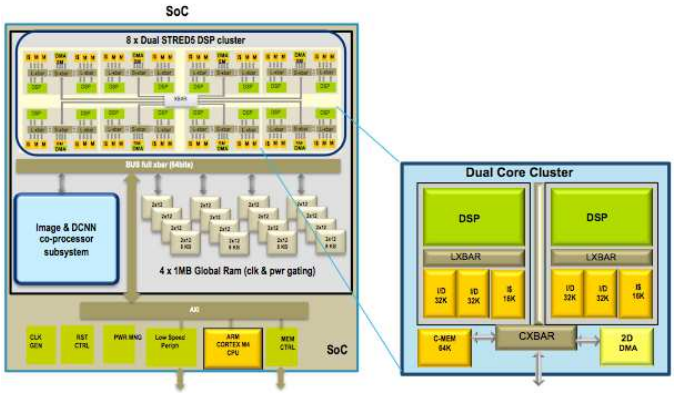

Figure 1: [a]Orlando high level system architecture on the left

[b] a single DSP cluster shown on the right

In order to validate the performances of the proposed parallel implementation, we compare it with the sequential execution model generated by an automated tool-chain. This is done by integrating our implementation with *VIVI-AI*, a ST tool-chain based on Python that parses a generic pre-trained DCNN, exported by an off-the shelf training tool. *VIVI-AI* generates automatically the ANSI C code, for the DCNN, which can be compiled and executed on a target platform (such as an STM32 microcontroller or an Orlando SoC). *VIVI-AI* can accept inputs from many of the most popular deep learning toolboxes such as Keras, Lasagne, Caffe, ConvNetJS.

As shown in Figure 2, it integrates three main components: 1) the Neural Network (NN) Exporter, the NN Mapper, and the NN Layers Library. The NN Exporter is fed with the pre-trained (either in Keras, Lasagne, or Caffe file format) DCNN network to generate a Framework Independent Neural Network Representation (INNR). The NN Mapper uses the INNR to output C code generated for the specific execution target and compiled with the application project. Internally, a graph is produced that represents the structure of the input DCNN in order to search optimization opportunities.

Finally the NN Layers Library is a module which contains all the supported NN layer optimizised implementations (Conv2D, Dense, etc.).

These functions are optimized in a target device dependent way to better and fully exploit the specific computing capabilities of the hardware resources (specific instruction and data buses, prefetch buffers, I and D caches, fast datapaths etc).

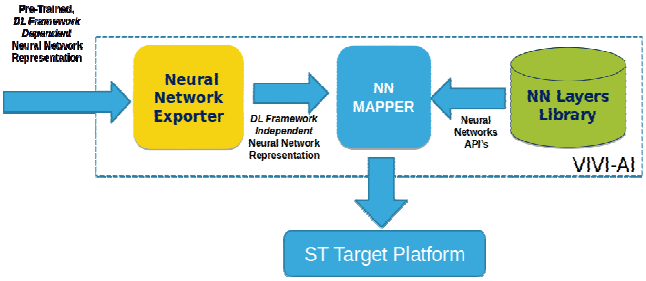

Figure 2: VIVI-AI Tool Architecture Overview

II. RELATED WORK

The contribution that this paper brings, consists in enhancing these tools by developing a specific scheme for parallelizing convolutions tailored to the Orlando architecture. To achieve that, we considered how the convolutions traverse the tensor structures during their processing. For example, the work in [2] describes how to partition an image into smaller parts and dispatch sub-images to the nodes of a cluster. Each node will then perform the image convolution on its own sub-image.

Other works present efficient parallel implementations of 2D convolutions demanding high performance computing power [3]. These works target cache based multi-CPU architectures. Unfortunately, those approaches cannot be easily extended to scratch-pad based multiprocessors such as the Orlando SoC.

Our implementation applies the work done in [2] to Orlando. This is done by considering the processing elements no longer as cluster nodes, but as individual processors. Moreover, it extends the work done in [2] by letting different processors to compute different set of filters. We delegate to the NN Mapper the choices for the configurations needed to parallelize 2D convolutions. Depending on the NN Mapper choice, different cores will compute different regions of the feature maps at the output of each NN layer. In this manner the NN Mapper can achieve what is called Inter Output Parallelism and Inter Kernel Parallelism [4]. The proposed method is experimentally tuned and it's indeed Orlando-specific. However, it can be applied to similar architectures if they are made of multiple cores and data-flow acceleration fabrics.

III. PROPOSED METHODOLOGY

In this section, we present the intra-layer approach that parallelizes convolution operations on tensors. A processing node (the DSP #0) acts as a master and at runtime dispatches parallel tasks to other DSPs. This is done by using a fork-join model. The approach considers the specific memory hierarchy of Orlando SoC, which has different latencies for different type of memories (as shown in Figure 3) and bus contention caused by concurrent accesses to the same bank.

A. High-level mapping

The master node (DSP #0) distributes chunks of input volume of activations to other nodes (DSPs). Thus, we decompose the input volume into sub-images and the master node distributes them to the P nodes.

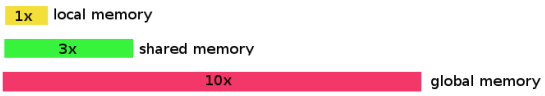

Figure 3: comparative memory latencies per each word access

Each node is responsible for a sub-image of sizes *h*w* and it performs the local convolution with *d* filters *k*k* for the layer. According to the sub-division described, our splitting policy requires three parameters for each 2D convolutional layer: the height and the width of the sub-image and the number of filters computed by each single DSP.

Depending on sub-image dimensions *(h,w)*, some parallelepipeds can have dimensions *(s, t, z)* with $s \neq h$ or / and $t \neq w$. This may happen because *w* (and *h*) is not in general divisible by the width (and height) of the input volume.

Our model also needs to know in advance how much memory need to be reserved for storing the filter coefficients. Since they usually can occupy large amounts of memory, we introduce a fourth parameter called *nkerns,* which represents the number of maximum concurrently computable filters by each node. If a DSP performs the local convolution using *d* filters, several tasks (called feature_batch) will be performed. A feature_batch is composed by $\lceil d/nkerns \rceil$ tasks.

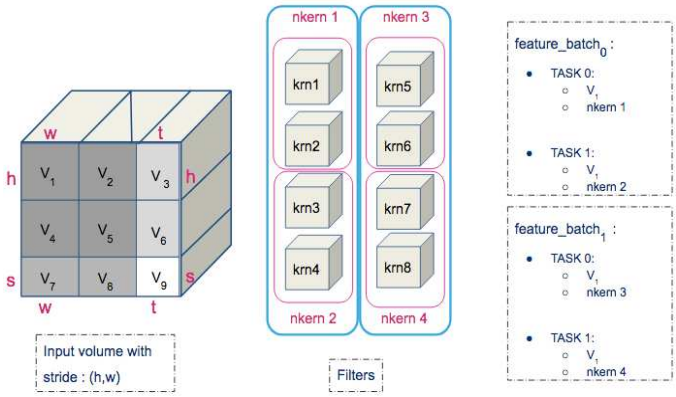

Figure 4: High-Level view of the proposed parallel splitting-policy

As shown in Figure 4, a single task included in a feature_batch consists of the computation of nkerns filters and one sub-image. For simplicity, in the example, we selected horizontal and vertical stride, respectively equal to w and h, to avoid sub images overlap. It could happen that $nkerns \nmid d$, in that case the last task consists of the computation of residual filters: *d mod nkerns.*

B. DSPs mapping

Orlando has a complex hierarchical memory and each of its components has different access time. For our purpose, the cluster common memory will not be used to store local filters and sub-images. This is because they are intended to contain ANSI C structures that are dynamically allocated by the master. These structures are required in order to have all the information needed to carry out a single local convolution by a particular DSP. Examples of such information are the shape of the sub-image, the shape of the kernel filters, and so on. The activations and the weights of the entire convolution are conveniently located in the 4MBytes on chip memory. To enable slaves to complete autonomously the assigned feature_batches, the DSP #0 at runtime will pass to other DSPs all the information needed for the local convolutions. For each task defined in the assigned feature_batch, each DSP will load first the corresponding nkerns filters. Then, the DSP will perform the local convolution between each sliding-window contained in the assigned sub-image and the previously loaded nkerns filters. Later on, the DSP will write the results into the global memory. To limit power consumption and achieve high throughput, an intra-task dual-buffering technique is used. We implemented it as follows: while a DSP is executing the convolution code of a sliding-window belonging to the sub-image of a certain task and the related nkerns set of filters, the data of the next sliding-window is loaded using the assigned 2D DMA into its local memory. In this way, the time of loading sliding-windows and the energy consumption are reduced thanks to the asynchronous usage of 2D DMA.

C. Optimizer

To verify performances achieved by our implementation on a complete DCNN, we integrated to the convolution layers, also other layers such as pooling and non-linear activation and VIVI-AI is instrumental to that. we change the implementation of a generic convolutional layer with a Orlando-specific one. This is achieved by substituting the sequential version (float-based) with the parallel one and finally evaluating its performance. In doing that, we also need to change the NN-Mapper in such a way that it makes optimizations based on the multi heterogeneous core architecture of Orlando SoC. More precisely, we developed the DCNN Optimizer related to Orlando (shown in Figure 5) to parallelize tasks among the DSPs and get the best mapping depending on user requirements (i.e. low power consumption or high performance). Therefore we reach a trade-off between the number of DSP cores used and the energy consumption achieved. Moreover, given P cores, the *(h,w,d)* configuration changes accordingly for each layer in order to minimize the execution time of the DCNN: i.e. the optimal *(h,w,d)* configuration with P cores might be different respect to the one with N cores (being $N \neq P$). Looking at the experiments, we noticed that the computation represents a bottleneck; this is due to how float-emulated MACs are executed on Orlando. For this reason, the optimizer will focus on how to obtain a balanced workload among the nodes. To achieve that, we propose an heuristic approach to identify a Pareto optimal solution that considers all the application-specific constraints. Supposing that *P* is the number of cores, it works only with 2D Convolutions that fulfill the following requirement: $\exists h,w,d\ (V \cdot K = P)$, where *V* is the number of the sub-images that are generated by splitting the input volume in sub-images having all (h,w) dimensions and K is the number of filters that are generated by splitting all filters in sets composed by d filters. In this way, we avoid the presence of the residuals.

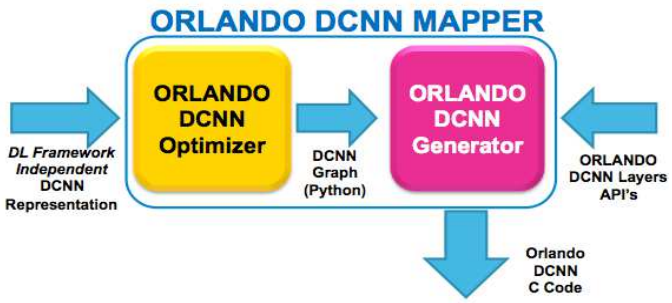

Figure 5: High-Level view of Orlando DCNN Mapper

The optimizer will choose one of the (*h,w,d*) configurations that satisfies the previous requirement. Obviously, it could happen that more than one solution is available. In that case, the optimizer will choose the one that opportunely maximizes *K* because kernels-wise parallelization gives better performance on the tested DCNNs. Then, for a certain *K*, it could happen that multiple configurations are available. In this case, it choices the one that maximize *w* because our double buffering implementation performs better with horizontal sub-images.

IV. EXPERIMENTAL RESULTS

The proposed methodology was tested on CifarNet, a low complex neural network designed to address the CIFAR-10 classification problem (10 objects categories, 60,000 color images of 32×32 RGB pixels) [5]. CifarNet (~10^5 parameters) is composed by three convolution layers interspersed by ReLU activation and max-pooling layers, followed by a fully-connected layer at the end of it. This network was introduced in [6] and re-implemented in the open source Caffe project [7]. As shown in Table 1, the speed-up observed is almost linear, despite the fact that only three convolutional layers were parallelized. The other layers were kept sequential. Considering that the 99.3 % of the execution time is due to convolutions, the maximum speed-up on a 16 DSP cores is limited to 14.5, accordingly to Ahmdal's law. This suggests that our implementation is extremely optimized in terms of parallelization. However, the average execution time is high because of the float emulation on Orlando's DSPs given that VIVI-AI does not support (yet) float to fixed-point layers.

V. CONCLUSIONS

In this paper, an efficient implementation of a task-based parallelized convolutions targeting Orlando was proposed and integrated into *VIVI-AI* tool by using the CifarNet NN to test the performances. The experimental results proved that the proposed implementation was highly parallel and supporting any kind of DCNN provided that parameters and activations fit into the global memory. Unfortunately real-time performances on DCNNs deeper than CifarNet were not achieved. This is due to multiple factors such as the lack of float to fixed-point conversion. Future work will address these specific weaknesses along with the HW accelerators mapping.

	(h,w,d)	**DSPs**	**mean ET [ms]**	**SU**
conv 1 conv 2 conv 3	(36,36,16) (20,20,20) (12,12,20)	1	1094	1x
conv 1 conv 2 conv 3	(36,36,8) (20,20,10) (12,12,10)	2	542	1.99x
conv 1 conv 2 conv 3	(36,36,4) (20,20,5) (12,12,5)	4	275	3.93x
conv 1 conv 2 conv 3	(36,36,2) (12,20,5) (8,12,5)	8	148	7.30x
conv 1 conv 2 conv 3	(36,36,1) (8,20,5) (6,12,5)	16	76	14.21x

Table. 1. Speed-up and mean execution time over CifarNet inference based on the average of 10 inputs. The results were obtained on Orlando at 800MHz with single precision float-emulated MACs (without SIMD instructions). The configurations were chosen by the optimizer.

VI. REFERENCES

[1] Giuseppe Desoli, et al. "The Orlando Project: A 28 nm FD-SOI Low Memory Embedded Neural Network ASIC", in (Springer, 2016), pp. 217—227

[2] Montero-Gonzalez, Rafael J., et al. "Performance study of software AER-based convolutions on a parallel supercomputer." *International Work-Conference on Artificial Neural Networks*. Springer, Berlin, Heidelberg, 2011.

[3] Kim, Cheong Ghil, Jeom Goo Kim, and Do Hyeon Lee. "Optimizing image processing on multi-core CPUs with Intel parallel programming technologies." *Multimedia tools and applications* 68.2 (2014): 237-251.

[4] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks", in Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific (, 2016), pp. 575--580.

[5] Alex Krizhevsky and Geoffrey Hinton, "Learning multiple layers of features from tiny images", (2009).

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, Imagenet classification with deep convolutional neural networks, (2012).

[7] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In arXiv, 2014.