Improving Deep Learning with a customizable GPU-like FPGA-based accelerator

Mirko Gagliardi, Edoardo Fusella, Alessandro Cilardo
Department of Electrical Engineering and Information Technologies, University of Naples Federico II
via Claudio 21, 80125 Napoli, Italy, Email: mirko.gagliardi@unina.it

***Abstract*—An ever increasing number of challenging applications are being approached using Deep Learning, obtaining impressive results in a variety of different domains. However, state-of-the-art accuracy requires deep neural networks with a larger number of layers and a huge number of different filters with millions of weights. GPU- and FPGA-based architectures have been proposed as a possible solution for facing this enormous demand of computing resources. In this paper, we investigate the adoption of different architectural features, i.e. SIMD paradigm, multithreading, and non-coherent on-chip memory for Deep Learning oriented FPGA-based accelerator designs. Experimental results on a Xilinx Virtex-7 FPGA show that the SIMD paradigm and multithreading can lead to an improvement in the execution time up to $5\times$ and $3.5\times$, respectively. A further enhancement up to $1.75\times$ can be obtained using a non-coherent on-chip memory.**

I. Introduction and Related Work

The emerging wave of the Big Data [1] is paving the way for the widespread adoption of Deep Learning techniques in diverse application domains including image recognition [2], sound processing [3], medical systems [4], gaming [5], and others. However, despite the huge potential of Deep Learning, most of these algorithms rely on a large number of performance-hungry convolutions limiting the usability of these techniques. In addition, Deep Neural Networks (DNNs) require a training phase that is a very compute intensive task. For instance, training a popular architecture like, e.g. GoogLeNet [6], can easily take several days on a standard GPU. Because of these requirements, the applicability of Deep Learning is becoming increasingly performance- or power-constrained.

Not surprisingly, the industry and academia are continuously introducing new architectures dictating the evolution of Deep Learning techniques. First-generation solutions consist of large-scale distributed systems comprised of tens of thousands of CPU cores [7]. However, the growing demand for high-parallel energy-efficient architectures has led to an increasing interest in GPUs and FPGAs [8], [9], [10]. For example, many entries in the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [11] use GPUs and FPGAs to implement DNNs. FPGAs are an attractive alternative and provide an intermediate point between Application-Specific Integrated Circuits (ASIC) and standard GPUs, enabling higher efficiency even compared to high-end GPUs [12]. In addition, the higher flexibility allows their use in different compute problems. There is thus a tradeoff between power-hungry high-performance GPUs and energy-efficient application-specific solutions.

In this paper we investigate the adoption of different architectural features, i.e. SIMD paradigm, multithreading, and non-coherent on-chip memory for Deep Learning oriented FPGA-based accelerator designs. We designed and implemented a customizable GPU-like SIMD architecture as a solution to support architecture-level exploration for Deep Learning oriented systems. Architectural customization plays a key role, as it enables unprecedented levels of resource-efficiency compared to GPUs. The accelerator was synthesized into a Xilinx Virtex-7 2000T XC7V2000T FPGA chip. Experimental results show that such a customization leads to significant improvements over non-customized architectures.

II. GPU-like architecture

This work relies on an experimental platform, called nu+, providing a parameterizable GPU-like architecture inspired by modern GPUs, yet exposing full customization capabilities for architectural exploration. The heart of the platform is a RISC in-order core oriented to highly data-parallel kernels with a lightweight control infrastructure, shown in Figure 1. Most of its resources are dedicated to computation-intensive operations on massive datasets. Such accelerator blends together a hardware multithreading paradigm with a vector processor model. Each hardware thread has private internal resources such as PC, register file, and control registers along with a private memory stack, although all threads share the same compute units, L1 cache and an on-chip non-coherent memory. The thread control unit implements an interleaved multithreading scheduling in a fine-grain way. An internal round robin arbiter issues instructions for different threads in a fair mode after every cycle with a low architectural impact. Execution datapaths and register files are designed to exploit data-level parallelism. The architecture implements an instruction set containing instructions that operate on arrays of data. Computational units are organized in hardware vector lanes, with each scalar operator being instantiated N times. Dually, each thread is equipped with a vectorial register file, where each register can store up to N scalar data in order to satisfy the execution pipeline data throughput. Such a data parallelism allows each thread to perform SIMD operations on N independent data simultaneously.

The proposed GPU-like architecture has an n-way set-associative write-back L1 cache strictly coupled with a light

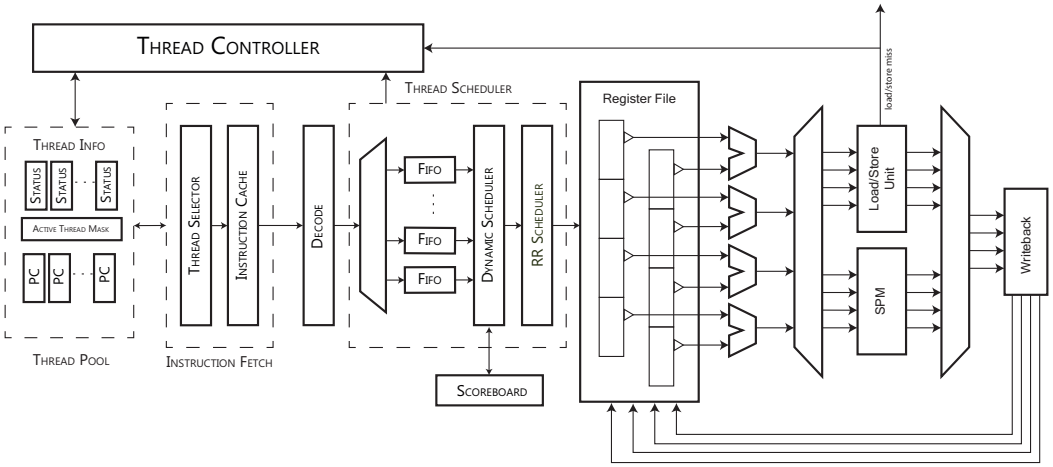

Fig. 1. Simplified overview of the developed GPU-like accelerator. This figure highlights both data and thread level parallelism. Beside memory configuration parameters, this architecture has different customization levels, such as the number of hardware lanes and the number of hardware threads implemented.

cache controller which implements a simple valid/invalid coherence mechanism. Such cache controller handles misses and memory transactions and it also provides both request serialization and merging mechanisms in order to correctly manage concurrent requests from different threads. The cache line width matches the internal hardware lanes capability, thus a read memory request loads N scalar data from main memory and stores them into a vectorial register at once, minimizing requests and exploiting the internal parallelism of the GPU-like accelerator.

As in general purpose platforms, the performance of custom accelerators is also critically dependent on data movement and memory accesses. In that respect, hardware coherence mechanisms introduce both architectural and data management overheads, which are not always necessary in some applications. Many modern parallel architectures utilize fast non-coherent on-chip memories, called *scratchpad memories* (SPMs). Since NVIDIA Fermi family, GPUs are equipped with this kind of memories, that are intensively used to facilitate communication across threads and to store partial outputs or temporary data that are not requested to be synchronized back into main memory.

The nu+ core supports such kind of high-throughput non-coherent scratchpad memory, which is divided in a parameterized number of banks based on a user-configurable mapping function in order to support multiple memory accesses. The scratchpad memory is organized in multiple independently-accessible memory banks providing a high data access parallelism. Therefore, if all memory accesses request data mapped to different banks, they can be handled in parallel. However, when multiple requests are made for data within the same bank, conflicts occur and a resolution logic handles and serializes each request resulting in a significantly performance loss. In fact, whenever an n-way conflict is detected, such a serialization logic notifies to the GPU-like accelerator control logic that the memory is not able to receive any further request, then it splits the conflicting requests into n conflict-free sub-requests issued serially in the next n cycles.

The implemented architecture comes with a toolchain based on the LLVM project and includes a custom version of the Clang frontend and a native nu+ backend. The Clang frontend allows users to compile traditional C/C++ source code in a fast way and with a low memory usage. On the other hand, the toolchain is deeply customized for exploiting the core internal data parallelism and reaching the maximum throughput. The compiler has a complete vision of the SIMD nature of the datapath. It supports custom vector types, thus standard arithmetic and bitwise operators are available for both scalar and vector operations. Furthermore, the custom version of Clang supports ad-hoc builtin functions that are required to fully exploit target specific features, such as thread synchronization and special SIMD operations.

Combining a non-coherent on-chip memory approach, high data parallelism, and a fine-grain thread control, this GPU-like accelerator provides a significant speed-up in compute-intensive and data demanding workloads.

III. Convolution algorithm

Deep Learning is a class of machine learning algorithms using convolutional neural networks (CNN) which are inspired by the behavior of optic nerves. Deep Learning gives state-of-the-art accuracy for many computer vision tasks, such as image classification and image search engine in data centers.

CNN employs a feedforward process for recognition and a backward path for training. Consequently a typical CNN is composed of multiple computation layers, and the output y is the sum of multiple different convolutions between the input x and the filter k:

$$y[n] = x[n] \cdot k[n] = \sum_{k} x[n] \cdot k[n-k]$$

Our work focuses on the exploration of different architectural features in a custom GPU-like accelerator targeted at convolution operations. In fact, these account for over 90%

of the processing in CNNs for both inference/testing and training [13].

The pseudo code of a convolution with a $K \times K$ filter with no stride, bi-dimensional input and output matrices, respectively of $N \times N$ and $M \times M$ where $M = N - K$, can be written as in the following listing:

```
for(row = 0; row < M; row++)
 for(col = 0; col < M; col++)
   for(krow = 0; krow < K; krow++)
     for(kcol = 0; kcol < K; kcol++)
       y[row][col] += k[krow][kcol] *
                      x[row + krow][col + kcol];
```

This scalar single-thread version of the convolution algorithm has been adapted to our target architecture exploiting both thread and data level parallelism. Output matrix row calculations are equally spanned across all threads, i.e., for each thread, the outer loop starts with the thread ID (*thid*) and increments by the number of threads (*thnumb*). On the other hand, both input and output matrices are organized in target specific vector types, becoming vectors of vectors. Such an organization results in a distribution of the M column partial results on the M hardware lanes; each thread calculates M partial results every cycle. The vectorization makes the second cycle unnecessary. The inner cycle, however, scrolls the input matrix in the scalar version. This can be replaced by a vectorial shift operation supported by the target architecture on the input row, which shifts each scalar element inside the hardware vector by n positions. The resulting pseudo code of the algorithm optimized for a GPU-like accelerator, can be written as in the following listing:

```
for(row = thid; row < M; row += thnumb)
  for(krow = 0; krow < K; krow++)
    for(kcol = 0; kcol < K; kcol++) {
      y[row] += k[krow][kcol] *
                x[row + krow];
      x[row + krow] =  x[row+krow] << 1;
    }
```

IV. Evaluation

We carried out our experiments on a proFPGA MB-4M FPGA board by ProDesign, equipped with one Xilinx Virtex-7 2000T XC7V2000T FPGA. The GPU-like accelerator has been developed in SystemVerilog hardware description language (HDL) and synthesized using the Vivado design suite provided by Xilinx. The design has been validated with the Verilator RTL simulator tool [14] and with an in-house event-driven cycle-accurate emulator. Finally, the convolution algorithm was written in *C* and compiled using our toolchain.

The convolutions were performed on 16×16, 32×32, and 64×64 input images with filter kernels of size between 3×3 and 7×7. We first carried out a set of experiments to assess speedup over a naive scalar single-thread implementation and estimate the performance boost of the SIMD paradigm. Figure 2 depicts the results. By sweeping the size of the input image from 16×16 to 64×64, we observe a great increase in the effective acceleration (up to $5\times$). This is due to the higher number of multiply-add operations that need to be performed. Unsurprisingly, small input images with filter kernels of size 7×7 have a reduced speedup due to the unbalanced sizes of the images and the filter causing a suboptimal use of the hardware lanes.

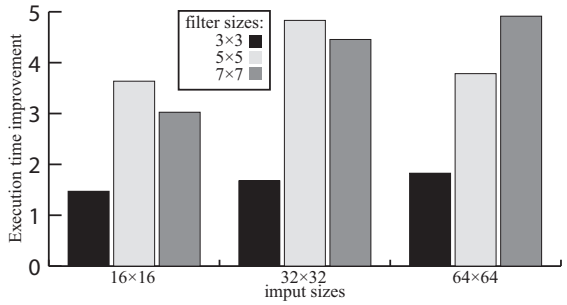

Fig. 2. Speedup over naive scalar single-thread implementation on 16×16, 32×32, and 64×64 input images with 3×3, 5×5, and 7×7 filter kernels.

Then, in a second set of experiments we evaluated the benefits of using multithreading. Figure 3 shows the speedup over a single-thread implementation. Two threads ensure a speedup between $1.3\times$ and $2\times$, while a higher number of threads leads to better performance (up to $3.5\times$). However, this trend is not constant since in case of small images and/or small filters, a higher number of threads may be useless. This is because hardware multithreading involves some overhead for handling the different stacks (one for each thread) and for thread scheduling and synchronization. For instance, in case of a 16×16 input image and filters with a size of 3×3 and 7×7, the optimum number of threads is respectively two and six. This is because convolutions performed on a 16×16 input image with filter kernels of size 7×7 require 2.6 more arithmetic operations than in case of 3×3 filters.

Finally, in the last set of experiments, we evaluated the benefits of using the scratchpad memory. The results in terms of speedup over an accelerator with a standard memory subsystem are summarized in Figure 4. In case of smaller filters, we observe better results since there is a lower need to swap data between the scratchpad and the main memory. In general, the achieved speedup scales up with the number of threads up to $1.75\times$. This is due to the higher efficiency of the scratchpad memory, which does not implement the coherence functionalities of traditional cache memories, as well as the lower number of cache misses when using the scratchpad memory (up to 30%).

V. Conclusions

In this work, we investigated various architectural features for the definition of an innovative GPU-like accelerator targeted at Deep Learning. In particular, we evaluated how those features impact the performance of convolutions, the dominant operation in Deep Learning applications. Thread level parallelism and SIMD operation achieve a great increase in the acceleration efficiency reaching respectively a speed-up of $3.5\times$ and $5\times$ over a scalar single-thread implementation of

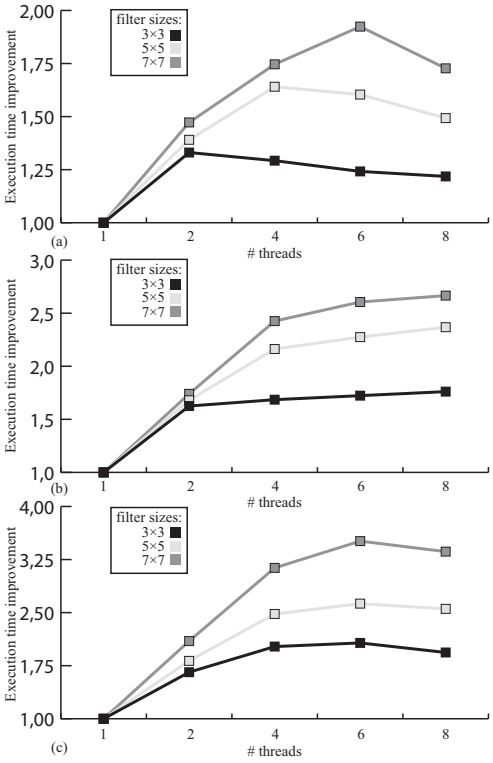

Fig. 3. Speedup over single-thread implementation when varying the number of threads on 16×16, 32×32, and 64×64 input images with 3×3, 5×5, and 7×7 filter kernels.

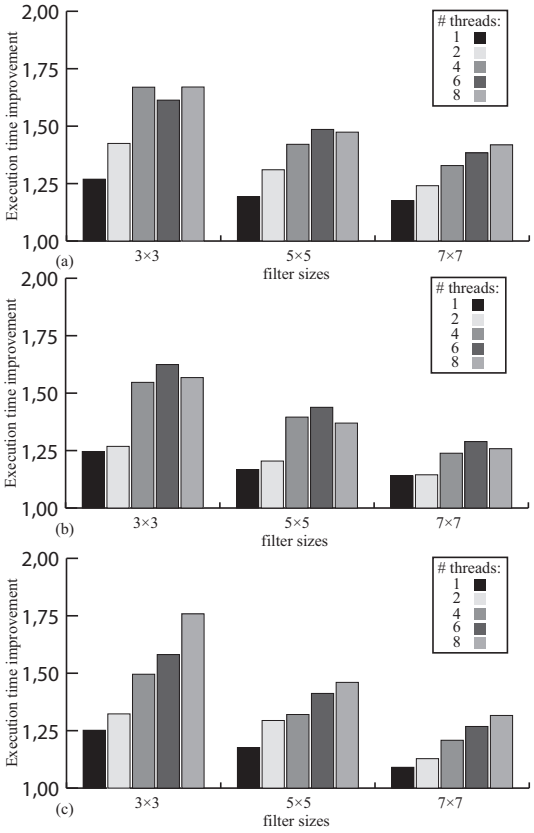

Fig. 4. The speedup achieved using scratchpad memory when varying the number of threads on 16×16, 32×32, and 64×64 input images with 3×3, 5×5, and 7×7 filter kernels.

the convolution algorithm. The use of non-coherent on-chip memory can further enhance performance to some extent, due to the increased locality and lower number of cache misses. In conclusions, the findings of our work may be particularly impactful for driving the long-term evolution of current accelerator architectures towards improved specialization and workload-specific customizability.

ACKNOWLEDGMENTS

This work is supported by the European Commission in the framework of the H2020-FETHPC-2014 project n. 671668 - MANGO: exploring Manycore Architectures for Next-GeneratiOn HPC systems.

REFERENCES

[1] K. Kambatla *et al.*, "Trends in big data analytics," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2561–2573, 2014.

[2] K. He *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] J. Salamon and J. P. Bello, "Deep convolutional neural networks and data augmentation for environmental sound classification," *IEEE Signal Processing Letters*, vol. 24, no. 3, pp. 279–283, 2017.

[4] A. Esteva *et al.*, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, p. 115, 2017.

[5] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[6] C. Szegedy *et al.*, "Going deeper with convolutions." Cvpr, 2015.

[7] J. Dean *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[8] Y. Chen *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[9] C. Farabet *et al.*, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 2011, pp. 109–116.

[10] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[11] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[12] C. Murphy and Y. Fu, "Xilinx all programmable devices: A superior platform for compute-intensive systems," *Xilinx White Paper*, 2017.

[13] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[14] W. Snyder, P. Wasson, and D. Galbi, "Verilator-convert verilog code to c++/systemc," 2012.