CNN-on-AWS: Efficient Allocation of Multi-Kernel Applications on Multi-FPGA Platforms

Junnan Shan, *Student Member, IEEE,* Mihai T. Lazarescu, *Senior Member, IEEE,* Jordi Cortadella, *Fellow, IEEE,* Luciano Lavagno, *Senior Member, IEEE,* and Mario R. Casu, *Senior Member, IEEE*

***Abstract*—Multi-FPGA platforms, like Amazon AWS F1, can run in the cloud multi-kernel pipelined applications, like Convolutional Neural Networks (CNNs), with excellent performance and lower energy consumption than CPUs or GPUs. We propose a method to efficiently map these applications on multi-FPGA platforms to maximize the application throughput. Our methodology finds, for the given resources, the optimal number of parallel instances of each kernel in the pipeline and their allocation to one or more among the available FPGAs. We obtain this by formulating and solving a mixed-integer, non-linear optimization problem, in which we model the performance of each component and the duration of the phases in which the accelerated computation can be split into, namely: 1) data transfer from a host CPU to the DDR memory of each FPGA, 2) data transfer from FPGA DDR to FPGA on-chip memory, 3) kernel computation on the FPGA, 4) data transfer from FPGA on-chip memory to FPGA DDR, 5) data transfer from FPGA DDR to host. Finding the optimal solution using a Mixed-Integer Non-Linear Programming (MINLP) solver is often highly inefficient. Hence, we provide a fast heuristic method that according to our experiments can be much more efficient than the MINLP solver and finds comparable results. For larger problems (more CNN layers), our heuristic method can quickly find (several thousand times faster) much better solutions than the MINLP solver, even if we run the latter for a very long time.**

***Index Terms*—CNNs, multi-FPGA, allocation, optimization.**

I. Introduction

CONVOLUTIONAL Neural Networks (CNNs) achieved breakthrough results in many challenging artificial intelligence domains, such as image recognition, object detection, and natural language processing. To continuously improve these results approaching human abilities in a broad variety of domains, the CNNs depth increases, thus leading to Deep Neural Networks (DNNs). Application domains of these networks range from processing live data for traffic surveillance cameras, to identifying peoples in pictures, transcribing voice and analyzing text to perform "sentiment analysis" (for customer support or to improve user experience on social networks).

Most of these applications are run on datacenter-class servers, for which processing speed and energy consumption are primary concerns. For these reasons, CPU- and GPU-based platforms are poorly suited and increase operating costs. ASICs can provide the best energy efficiency, but the continuous evolution of DNNs requires flexible ASICs, such as the Google TPU [1], which are, however, less efficient than theory would predict.

J. Shan, M.T. Lazarescu, L. Lavagno and M.R. Casu are with the Department of Electronics and Telecommunications, Politecnico di Torino, I-10129 Torino, Italy, e-mail: mario.casu@polito.it.

J. Cortadella is with the Computer Science Department, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain, e-mail: jordi.cortadella@upc.edu

Manuscript received Month XX, 20ZZ; revised Month YY, 20ZZ.

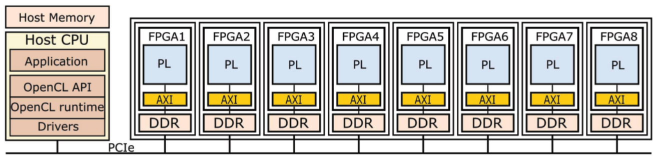


Fig. 1: Architecture of the Amazon Web Service (AWS) F1 instance.

FPGAs are a promising option for CNN and DNN acceleration in datacenters, offering energy efficiency coupled with full re-programmability and configurability for both datapath and memory architecture. This allows one to tailor the architecture to the application to a much deeper extent than either CPU/GPU platforms or relatively rigid domain-specific ASICs, like the Google TPU. For these reasons, cloud providers like Amazon Web Service (AWS) offer Virtual Machines coupled with multi-FPGA platforms to accelerate datacenter applications with GPU-like performance, but consuming less energy.

Most past work addressed CNN acceleration on a single FPGA. However, since network depth and complexity increase, single-FPGA designs cannot always meet performance requirements and would benefit from multi-FPGA implementations. In this work, we address the problem of optimizing the implementation of these applications on multi-FPGA platforms in such a way to maximize their throughput.

We run our experiments on an AWS F1 instance. As shown in Fig. 1, it has eight Xilinx UltraScale+ FPGAs, each equipped with local DDR DRAM and connected via the PCIexpress (PCIe) bus to an x86 host CPU. The role of the host CPU is to orchestrate the execution of the applications on the FPGAs and allow them to communicate via PCIe.

We use an OpenCL-like (but not OpenCL-limited) execution model, in which an application is typically (but not always, as we briefly discuss later) a linear task-level pipeline of kernels. Fig. 2 is an example of a **K**-stage kernel pipeline. In the context of CNNs and DNNs, the kernels correspond to layers: convolutional, max-pooling, normalization, etc. Each kernel is mapped to one or several independent Compute Units (CUs), depending on the level of parallelism required for that kernel, on one or more FPGAs. In Fig. 2, each pipeline stage is mapped to a specific number of CUs $(N_1, N_2, \ldots, N_k)$.

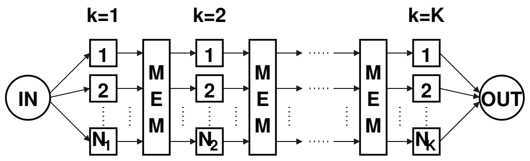


Fig. 2: Example of a **K**-stage kernel pipeline.

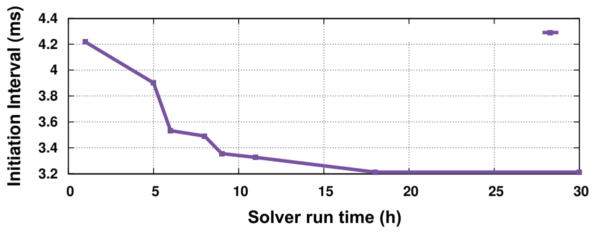


Fig. 3: Slow progress of the MINLP solver while searching for the optimum allocation of a CNN application.

The CUs are implemented in the FPGAs using a High-Level Synthesis (HLS) flow. The CUs are optimized using loop tiling and permutation of nested loops to reduce data dependencies and increase parallelism [2]. Each CU executes loops, which can be unrolled and pipelined with HLS to further increase the performance. Kernels communicate between them and with the host CPU via large buffers allocated in the external DRAM, i.e. the MEM blocks in Fig. 2.

The CU-level parallelism can be arbitrarily increased via replication. This computational model is also supported by C++-based synthesis tools[1], and fits very well many data-center applications, like CNNs or other Neural Networks and Machine Learning algorithms, databases, video encoding and decoding algorithms, and so on.

Optimizing the global throughput of a task-level pipelined application, however, is not a trivial task. A designer needs to:

- balance the number of CUs of each kernel, knowing that in an OpenCL-style task-level pipeline, the application throughput is the inverse of the latency of the slowest stage of the pipeline;
- allocate the CUs in the FPGAs trying to maximize communication locality;
- meet the FPGA constraints on memory bandwidth and resources: Look-Up Tables (LUTs), Block RAMs (BRAMs), Flip-Flops (FFs), and Digital Signal Processing (DSP) blocks.

Indeed, the optimization problem can be mathematically formulated as a complex Mixed-Integer Non-Linear Problem (MINLP), which turns out to be particularly hard to solve using commercial or academic solvers. As an example, Fig. 3 shows the slow progress of the Couenne solver [3] when optimizing the Initiation Interval (II), which is the inverse of the pipeline throughput, of the YOLO CNN [4] on three FPGAs with a specific resource utilization constraint (namely 45% target maximum resource usage, to ensure good routability and fast clock frequency).

To accelerate the optimization process, we propose a fast heuristic that not only returns the solution in a matter of seconds, instead of several hours or days run time of the MINLP solver, but often offers better results than those returned by the solver when its run time is limited for practical reasons.

In our previous work [5] we did not model the data transfer time between the CPU and the FPGAs. Here, instead, we consider both that time and the fact that the communication between kernels mapped to the same FPGA can occur within a board, thus avoiding costly inter-board data transfers through the host CPU (the AWS platform does not yet offer direct inter-FPGA transfers via PCIe links). For the execution phase, we separate the DDR access time from the computation time to improve the model accuracy. We also consider the effects of clock frequency reduction when the resource utilization increases.

We improve also our heuristics, in order to tackle the more complex performance and cost model, and increase the number of CNN benchmarks for which we show results, now including AlexNet [6], VGG [7], YOLO [4] and ResNet [8]. Each of these networks consists of convolution layers, pooling layers (sometimes combined with the previous convolution layer for efficiency), and, only in AlexNet, normalization layers. Although we restrict our results to these benchmarks, our technique is completely general and applicable to any DNN or deep task-level pipelined application.

Our main contributions are:

- A mathematical model that covers the whole application execution, which consists of the following sequence:
  1) input data transfer time from the host CPU to the FPGA DDR memory, which is considered only if needed, i.e., if the data are bound to the first kernel in the pipeline, or to a kernel allocated on different FPGA(s) than the kernel that sends the data,
  2) data transfer time from the FPGA DDR memory to the FPGA on-chip memory,
  3) actual kernel computation,
  4) data transfer time from the FPGA on-chip memory to the FPGA DDR memory, and
  5) data transfer from the FPGA DDR memory to the host CPU, again which is considered only if needed, i.e., if the data come from the last kernel in the pipeline, or from a kernel allocated on different FPGA(s) than the kernel that receives the data.
- An implementation of the model suitable for being solved by a MINLP solver, which finds a solution that maximizes the *global execution throughput* by minimizing the II of the kernel pipeline, which is the product of the cycle count times the estimated clock period.
- A heuristic method that integrates Geometric Programming (GP) to relax the constraints of the exact model, followed by an efficient allocation algorithm that returns the number of compute units (CUs) for each kernel, and their allocation on various FPGAs.

Our article is organized as follows. In Section II, we discuss related works. In Section III, we present the problem formula-

[1]In fact, in this work we model our applications in C++ to better control loop handling during HLS, since the Xilinx OpenCL HLS front-end is not yet as developed as their C++ one.

tion, and discuss the proposed heuristic method in Section IV. In Section V, we present and discuss the experimental results. Section VI concludes the article and outlines opportunities for future work.

II. Related Work

The community interested in compilers for parallel architectures faced a similar problem when mapping streaming applications to multiprocessor systems and accelerators. In fact, in [9], the authors define three levels of parallelism (task, data, and pipeline) that we also exploit in our execution model (tasks are called "kernels", data parallelism is exploited both at the CU level and at the loop unrolling level within a CU, and the innermost loops within each CU are pipelined). However, their compiler is aimed at processors rather than FPGAs. Moreover, it only makes heuristic choices for allocation, while we first find an optimal non-integer solution using Geometric Programming, then we relax it, obtaining very good results.

In terms of FPGA implementation of DNNs, the research focus moved from single to multiple accelerators (i.e., the layers of a DNN) implemented on a single FPGA [10], [11], [12]. Even though in these works the use of FPGA resources and memory bandwidth are maximized, still single FPGA designs cannot deliver the performance of multi-FPGA platforms, which have recently attracted the interest of researchers.

In [13], the authors propose Multi-FPGA CNN acceleration by minimizing *independently* the latency of each kernel, while our goal is to maximize the application throughput. Their design space exploration is applied to each layer individually, which may oversize or undersize each layer with respect to the global balancing of the task-level pipeline. However, similar to our work, [13] also adopts an on-board data reuse scheme to minimize the external memory access time.

In [14], the pipeline stages are consecutive kernels allocated on a single FPGA and the throughput is optimized by balancing the workload and the FPGA resources. The II of the pipeline in [14] is by construction greater than in our work, and therefore the throughput lower, because the kernels of each group are executed sequentially within a single FPGA. The advantage of our method is that all kernels can work concurrently regardless their allocation in the FPGAs, since each kernel is a single stage of the pipeline. Moreover, in [14] the consecutive kernels are forced to be allocated on the same FPGA, while our model does not force that. Finally, [14] does not consider the frequency reduction due to routing congestion when the resource utilization increases, while we consider it.

Similar to our work, in [15] the authors first obtain a characterization of individual kernels, which then they use to feed a dynamic programming model that optimizes the way in which the network is partitioned into stages. Still, our model can obtain a better II for the same reason that it can outperform the results obtained by the method proposed in [14], namely that we do not restrict the distribution of CUs to FPGAs to be grouped by stages.

In [16], the authors focus on designing optimal pipelined CNNs on a set of heterogeneous FPGAs. The rationale is that different tasks in the pipeline are better suited to a specific type of FPGA. Our work is different from theirs in various aspects, of which the main three are as follows. First, we target an existing commercial Multi-FPGA platform (AWS), which consists of a set of homogeneous FPGAs, but our formulation can be adapted to heterogeneous FPGAs. Second, we do not force neighboring pipeline stages to be on the same FPGA, but we take into account the performance advantage of doing so to achieve a globally better solution. Third, to improve the solver efficiency, [16] provides an efficient BLAST algorithm using Dynamic Programming (DP), while we use a Geometric Programming solver and a heuristic allocator to improve the efficiency.

Finally, [17] and [18] propose to accelerate a lung cancer nodule segmentation algorithm on a multi-FPGA system. All these works maximize the application throughput using pipelined FPGA clusters, i.e., they force neighboring stages to be on the same FPGA, which may or may not be the best solution. Our work uses the layers of the DNNs as a more natural partition of the network into pipeline stages. Differently from previous works, we also consider an *estimated* clock frequency reduction due to routing, when FPGA resource usage increases.

III. Problem Formulation

We consider a multi-kernel application, like a CNN or DNN, as a set of K kernels organized as stages of a linear pipeline, i.e., $\{1, 2, \ldots, \mathsf{K}\}$ as in Fig. 2. However, unlike [14], [16], [17], [18] we do not limit the allocation to follow strictly this logical pipeline, because we do not force several adjacent kernels to be grouped as a single stage of the pipeline and be allocated on a single FPGA, although we can exploit this when advantageous. In CNNs and DNNs, the kernels are the convolutional, pooling, and normalization layers[2]. The workload of each kernel, say the kth stage, is assigned to N_k CUs that operate concurrently. We consider kernels that are inherently parallel and for which the execution time scales proportionally with the number N_k of CUs for that kernel[3].

Application throughput is the inverse of the pipeline initiation interval (II), which depends on the execution time of the slowest pipeline stage. To minimize II, we must find the optimal value of N_k and the CU allocation on multiple FPGAs under specific constraints. If we define $n_{k,f}$ as the number of CUs of kernel k on FPGA f, we have

$$N_k = \sum_{\mathsf{f}=1}^{\mathsf{F}} n_{k,f}, \tag{1}$$

where F is the number of available FPGAs (e.g., $\mathsf{F} = 8$ for the AWS F1.16xlarge).

By increasing N_k to decrease the execution time, one has to consider not only the FPGA resource limitations, but also the limited memory bandwidth. Indeed, the CUs fetch from the

[2]We merge some max-pooling layers with the previous convolutional layer whenever this allows us to optimize memory access. We do not implement the fully connected layers, since we are simply interested in showing a design methodology with a realistic use case, rather than benchmarking a full application.

[3]This "unlimited parallelizability" is a key reason for the success of modern DNN algorithms.

TABLE I: Constants (boldface) used in model equations.

Notation	Description
$\mathbf{K}$	number of kernels: $k \in \{1, \dots, \mathbf{K}\}$ used as index
$\mathbf{F}$	number of FPGAs: $f \in \{1, \dots, \mathbf{F}\}$ used as index
$\mathbf{C}_k$	Input constant data of kernel k
$\mathbf{B}_{H2F}$	bandwidth of the link between host and FPGAs
$\mathbf{DI}_k$	input data of k, not including constants
$\mathbf{B}_{F2H}$	bandwidth of the link between FPGAs and host
$\mathbf{DO}_k$	output data of k
$\boldsymbol{\delta}_k$	duplication factor for non-constant inputs
$\boldsymbol{\gamma}_k$	duplication factor for constant inputs
$\mathbf{r}_k$	number of AXI ports used by k only to read
$\mathbf{rw}_k$	number of AXI ports used by k both to read and write
$\mathbf{x}_k$	number of AXI ports used by k to read
$\mathbf{BDR}$	read bandwidth of local DDR
$\mathbf{w}_k$	number of AXI ports used by k only to write
$\mathbf{y}_k$	number of AXI ports used by k to write
$\mathbf{BDW}$	write bandwidth of local DDR
$\mathbf{TC1}_k$	worst-case computing time of kernel k when $N_k = 1$
$\mathbf{F1}_k$	clock frequency of kernel k when $N_k = 1$
$\mathbf{L1}_k$	latency (in clock periods) of kernel k when $N_k = 1$
$\mathbf{R}_t$	upper bound of usage for resource t in one FPGA: $t \in$ {BRAM, DSP, LUT, FF, AXI} used as index
$\mathbf{R}_{k,t}$	usage of resource t by kernel k in one FPGA

external DRAM the intermediate data and constants needed for their computation through AXI ports as shown in Fig. 1. We do not consider (yet) the possibility of streaming data directly between kernels, because it involves complex routing of data at runtime. We simply assume that if all the CUs of two adjacent kernels are on the same FPGA, then the host does not need to gather and scatter the data between them. This is a reasonable assumption for applications and platforms where the number of pipeline stages (i.e., kernels) is significantly larger than the number of FPGAs.

We formulate the optimization problem as follows:

$$\text{minimize} \quad II \tag{2}$$

$$\text{subject to}$$

$$N_k \geq 1, \quad \forall k \tag{3}$$

$$\sum_k n_{k,f} \mathbf{R}_{k,t} \leq \mathbf{R}_t, \quad \forall f, \ \forall t, \tag{4}$$

where the goal is to minimize II. Constraint (3) guarantees that each kernel is implemented with at least one CU. Constraint (4) defines an upper bound of resource utilization in each FPGA for all types t of FPGA resources, i.e., DSPs, BRAMs, Flip-Flops, LUTs, and AXI ports.

The problem difficulty stems from the complex dependencies between the II and the main optimization variables $n_{k,f}$, which will be thoroughly explained in the next subsections. In particular, the presence of integer variables and non-linear equations and constraints makes the problem a member of the Mixed-Integer Non-Linear Problem (MINLP) class.

All the constants and variables used in the model equations introduced in this and the following sections are reported in Table I and Table II, respectively. Note that we use bold typefaces for constants and regular typefaces for variables.

A. Modeling of application Initiation Interval (II)

We divide the execution time of each stage of the pipeline in three phases:

TABLE II: Variables (regular typeface) used in the model equations.

Notation	Description
$n_{k,f}$	CUs of kernel k allocated to FPGA f
N_k	sum of $n_{k,f}$ over all the F FPGAs
T_{h2f}	host-to-FPGA transfer time
T_{f2h}	FPGA-to-host transfer time
T_{exe}	Execution phase time
DI_{H2F}	total input data transferred in H2F phase
DI_D	total input data locally stored in DDR memories
a_k	binary, 1 if k's inputs are in DDR, 0 otherwise
$\alpha_{k,f}$	binary variable, 1 if k's CUs are in f
α_k	number of FPGAs in which k's CUs are spread
DO_{F2H}	total input data transferred in F2H phase
DO_D	total input data locally stored in DDR memories
b_k	binary, 1 if k's outputs are in DDR, 0 otherwise
$ET_{k,f}$	execution time of k in f
$TR_{k,f}$	reading time of k in f
$TC_{k,f}$	computing time of k in f
$TW_{k,f}$	writing time of k in f
dr_k	data read from DDR by each of k's CU
dw_k	data written to DDR by each of k's CU
BX_f	AXI bandwidth in f
NR_f	num. of AXI ports concurrently reading from f's DDR
$BR_{k,f}$	instantaneous read bandwidth of k's CU in f
NW_f	num. of AXI ports concurrently writing to f's DDR
$BW_{k,f}$	instantaneous write bandwidth of k's CU in f
L_k	latency (in clock periods) of kernel k for any N_k
$F_{k,f}$	clock frequency of kernel k in FPGA f
ψ	clock frequency degradation factor
R_f	resource usage metric for clock frequency computation
F_f	clock frequency of FPGA f
$\widehat{TC}_k$	computing time of k's CU in Geometric Programming (GP)
$\widehat{TC}$	maximum computing time among all the kernels in GP
II	initiation interval
$\widehat{N}_k$	total number of CUs of kernel k in GP
$\widehat{L}_k$	latency (in clock periods) of kernel k for any $\widehat{N}_k$

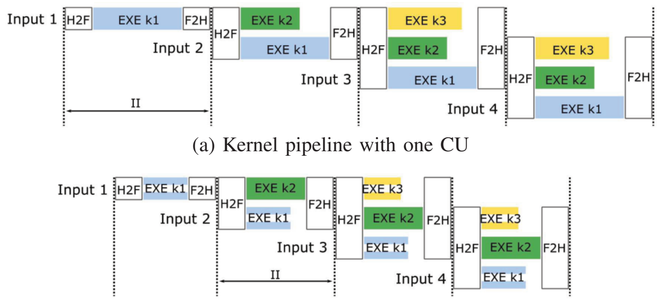
1) *Host-to-FPGA (H2F)* data transfer phase: the host transfers the input data from its own memory to the various DDR memories locally connected to the FPGAs. We denote the transfer time of this phase as T_{h2f}.
2) *Execute (EXE)* phase: all CUs fetch input data from the local DDR memory, perform the computation, and save the data back in the local DDR memory. The duration of this phase is T_{exe}, and it is the maximum among the execution times of the various kernels.
3) *FPGA-to-Host (F2H)* data transfer phase: the host transfers the output data from the local DDR memories to its own memory. We denote the transfer time as T_{f2h}.

Therefore, we can write

$$II = T_{h2f} + T_{exe} + T_{f2h}. \tag{5}$$

Note that if the three times were comparable, we could pipeline the three phases at the cost of double-buffering the DDR. We leave this further optimization for future work.

Fig. 4 shows an example of pipelined execution of three kernels. In Fig. 4(a), each kernel is implemented with one CU. In Fig. 4(b), kernels $K1$ and $K3$ use two CUs each, which leads to a significantly lower II. Note how the duration of the EXE phase is related to the maximum execution time among the various kernels. The kernels that determine this maximum



(a) Kernel pipeline with one CU

(b) Kernel pipeline with multiple CUs

Fig. 4: Initiation Interval (II) depends on the number of compute units of each kernel in a multi-kernel pipeline.

might change, depending on the number of CUs: in Fig. 4(a), $K1$ sets the II, while in Fig. 4(b) it is set by $K2$.

In the initialization phase, before pipeline inception, all constant data are transferred from the host to the DDR memories locally connected to the FPGAs. For example, in the CNNs these constant data are the weight and bias values. We define $\mathbf{C}_k$ to be the amount of constant data for each kernel. The duration of this transfer is not considered in the optimization, because it is typically small, since it occurs only once.

The modeling of the three phases is illustrated in the following sections.

B. Host-to-FPGA (H2F) phase

The duration of this phase is

$$T_{h2f} = \frac{DI_{H2F}}{\mathbf{B}_{H2F}}, \tag{6}$$

where DI_{H2F} is the total amount of transferred input data (in bytes) and $\mathbf{B}_{H2F}$ is the bandwidth of the link between the host and FPGAs (in GB/s), which is primarily the PCIe bus bandwidth (see Fig. 1). Note that DI_{H2F} *is not the total amount of input data for every kernel.* Part of the input data, which we denote as DI_D, is already stored in the local DDR and does not need to be transferred during H2F. This happens when *all the CUs of two adjacent stages of the pipeline (kernels $k-1$ and k) reside in the same FPGA*, therefore the output of kernel $k-1$, which is the input of kernel k, does not need to be transferred.

We model this using a binary variable, $a_k \in \{0, 1\}$, which denotes whether kernel k already has all its input data in the local DDR ($a_k = 1$) or not ($a_k = 0$):

$$a_k = \bigvee_{k>1,\forall f} \left((n_{k-1,f} = N_{k-1}) \wedge (n_{k,f} = N_k) \right). \tag{7}$$

Note that a_k is zero for the first kernel ($k = 1$), which always receives its input data from the host CPU. For the other kernels ($k > 1$), the logic expression (7) is true only if all the CUs of consecutive kernels ($k-1$ and k) are on the same FPGA.

The input data of kernel k, denoted as $\mathbf{DI}_k$, can be either in the local DDR or must be transferred from the host memory, but does not include the constant data which are in the local DDR after initialization. Thus, the part of the input data that is already in the local DDR, because it is produced by the previous kernel, is

$$DI_D = \sum_{k=1}^{\mathbf{K}} a_k \mathbf{DI}_k. \tag{8}$$

If kernel k does not already have its input data in local DDR, then it will receive $(1 - a_k)\mathbf{DI}_k$ data during H2F. Note that some networks like ResNet [8] violate the linear pipeline scheme of Fig. 3 and include branches that reconverge. In this case we can split the input data of one layer in two or more parts depending on how many branches reconverge to that layer. In terms of modeling, this requires a simple change of (8); in terms of implementation, this simply requires adding more memory buffers in DDR.

Each kernel k can have its CUs spread across multiple FPGAs. When they are spread, these data need to be duplicated[4]. Let us denote as α_k the number of different FPGAs in which the CUs of kernel k are spread. This is obtained as follows:

$$\alpha_{k,f} = \begin{cases} 1 & \text{if} \quad n_{k,f} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

$$\alpha_k = \sum_{f=1}^{\mathbf{F}} \alpha_{k,f}. \tag{10}$$

Finally, the total amount of data to be transferred during the H2F phase is

$$DI_{H2F} = \sum_{k=1}^{\mathbf{K}} \alpha_k (1 - a_k) \mathbf{DI}_k. \tag{11}$$

Fig. 5 illustrates an example of H2F phase with a hypothetical allocation of four kernels in three FPGAs. The constant data, $\mathbf{C}_1 - \mathbf{C}_4$, have been pre-transferred at initialization. Since kernels $K3$ and $K4$ are allocated to the same FPGA, the input data $\mathbf{DI}_4$ is not transferred during H2F, whereas $\mathbf{DI}_1$, $\mathbf{DI}_2$, and $\mathbf{DI}_3$ are all transferred. Note that it is necessary to transfer $\mathbf{DI}_2$ because not all CUs of $K1$ are allocated to the same FPGA as $K2$.

C. FPGA-to-Host (F2H) phase

Similar to the H2F phase, the duration of the F2H phase can be expressed as

$$T_{f2h} = \frac{DO_{F2H}}{\mathbf{B}_{F2H}}, \tag{12}$$

where $\mathbf{B}_{F2H}$ is the bandwidth and DO_{F2H} is the output data to be transferred to the host. Like before, a part of the output data remains in the local DDR, DO_D. To model this, we introduce another binary variable, b_k, for each kernel:

[4]We assume, for simplicity, that the host CPU only needs to know where kernel k is allocated and not which CUs are in each of the FPGAs where k is allocated. As a result, the host will simply duplicate the data transfer α_k times. As discussed above, a more precise model of data scattering and gathering is left to future work.

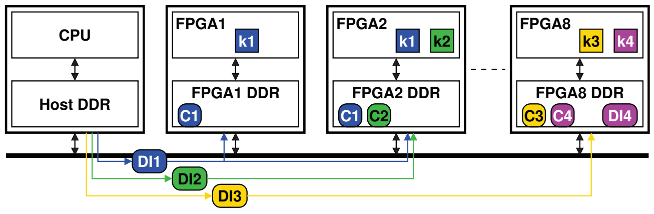


Fig. 5: Host-to-FPGA (H2F) example showing data transfer between host DDR and FPGA DDRs. $\mathbf{DI}_4$ is not transferred because kernels $K3$ and $K4$ are on the same FPGA. $\mathbf{DI}_2$ is transferred because parts of $K1$ are on different FPGA than $K2$.

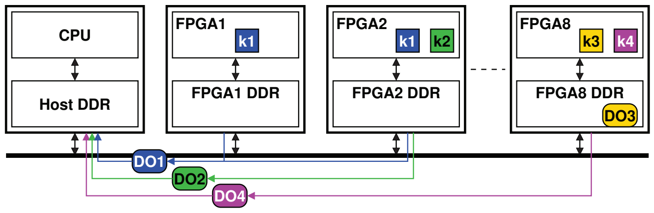


Fig. 6: FPGA-to-Host (F2H) example showing data transfer between FPGA DDRs and host DDR. $K3$ and $K4$ are on the same FPGA and $\mathbf{DO}_3$ does not need to be transferred.

$$b_k = \bigvee_{k<\mathbf{K},\forall f} \left((n_{k,f} = N_k) \wedge (n_{k+1,f} = N_{k+1})\right). \quad (13)$$

Note that b_k is zero for the last kernel ($k = \mathbf{K}$, which always transfers its output to the host CPU), and that for the kernels between 1 and $K-1$ its value is $b_k = a_{k+1}$.

If we define as $\mathbf{DO}_k$ the output data of kernel k (which can be either in the local DDR or be transferred to the host memory), we have

$$DO_D = \sum_{k=1}^{\mathbf{K}} b_k \mathbf{DO}_k \quad (14)$$

$$DO_{F2H} = \sum_{k=1}^{\mathbf{K}} (1-b_k)\mathbf{DO}_k. \quad (15)$$

Note that $\mathbf{DO}_k = \mathbf{DI}_{k+1}$ for kernels between 2 and $\mathbf{K}-1$.

Note also that, contrary to the input data, there is no output data duplication. Each CU, regardless of its allocation, contributes to a unique, non-duplicated fraction of the total output data $\mathbf{DO}_k$ of kernel k.

Fig. 6 shows the F2H phase of the same hypothetical allocation in Fig. 5. Since kernels $K3$ and $K4$ are in the same FPGA, the output data $\mathbf{DO}_3$ are not transferred during H2F, whereas $\mathbf{DO}_1$, $\mathbf{DO}_2$, and $\mathbf{DO}_4$ are all transferred.

This model does not force the CUs of a kernel, nor consecutive kernels, to be allocated on a single FPGA. However, grouping can reduce the data transfer time, which is part of the II. Hence, when the solver optimizes the II, it will implicitly try to group the CUs on a single FPGA. And for the same reason, consecutive kernels are also grouped together whenever possible.

D. Processing (EXE) phase

In comparison with our previous work [5], we improve the accuracy of the model by including the memory access time. This is obtained by dividing the execution time into three stages: reading data from DDR, performing computation and writing data to DDR.

The execution time of the CUs of kernel k located in FPGA f is $ET_{k,f}$, made of reading, computing, and writing times[5]:

$$ET_{k,f} = TR_{k,f} + TC_{k,f} + TW_{k,f}. \quad (16)$$

The duration of the EXE phase is obtained taking the maximum of all execution times:

$$T_{exe} = \max_{k,\,f} ET_{k,f} \quad (17)$$

The three times in (16) depend on the number of CUs of kernel k, N_k, as shown in the following.

1) Reading from local DDR: Let us define as dr_k the amount of total data that one CU of kernel k reads from the local DDR memory. These data include the input data, $\mathbf{DI}_k$, and the constant data, $\mathbf{C}_k$. If the workload is perfectly balanced among the CUs of a kernel, these data will be split in N_k chunks, and each CU will fetch one of these chunks. It is possible, however, that some data and/or some constants are duplicated. We introduce two factors to take into account the possible duplication of some of the data and the constants, $\boldsymbol{\delta}_k$ and $\boldsymbol{\gamma}_k$, respectively, such that we can express dr_k as follows:

$$dr_k = \frac{\boldsymbol{\delta}_k \mathbf{DI}_k + \boldsymbol{\gamma}_k \mathbf{C}_k}{N_k} + (1-\boldsymbol{\delta}_k)\mathbf{DI}_k + (1-\boldsymbol{\gamma}_k)\mathbf{C}_k, \quad (18)$$

where $0 \le \boldsymbol{\delta}_k \le 1$ and $0 \le \boldsymbol{\gamma}_k \le 1$. (18) captures the fact that not all input data scale with N_k and a residual amount of data needs to be fetched by all the CUs from local DDR even when $N_k \to \infty$. The two extreme values of $\boldsymbol{\delta}_k$ and $\boldsymbol{\gamma}_k$ capture the extreme cases of full duplication ($\boldsymbol{\delta}_k = \boldsymbol{\gamma}_k = 0$) and perfect scaling ($\boldsymbol{\delta}_k = \boldsymbol{\gamma}_k = 1$). The values of these constants can be obtained by profiling a few instances of the application with different CU allocations.

Each CU of kernel k accesses the local DDR through separate AXI ports, each with bandwidth BX_f. Note that the AXI bandwidth can be different in each FPGA (hence the f subscript) due to the specific clock frequency at which FPGA f is running. We also assume that all CUs start reading at the same time, and those that need less data or have the best memory bandwidth finish first, as we will discuss below.

The read bandwidth of the DDR connected to each FPGA is $\mathbf{BDR}$. This bandwidth is instantaneously shared among the NR_f *actively reading* AXI ports associated to the various kernels allocated to that FPGA. NR_f changes over time, due to the different finishing time. The instantaneous read bandwidth for each CU is therefore the minimum between

[5]Here we assume that reading, computing, and writing do not overlap, i.e. that task-level pipelining is not used inside the kernel to further optimize throughput at the expense of on-chip RAM usage. Including this aspect would require a simple modification of our model, using the max instead of the sum, which is not considered here.

the total AXI bandwidth used by the CU and the portion of DDR bandwidth that the CU receives:

$$BR_{k,f} = \mathbf{x}_k \cdot \min\left(BX_f, \frac{\mathbf{BDR}}{NR_f}\right), \tag{19}$$

where $\mathbf{x}_k$ is the number of AXI ports used in the reading phase. Some of these ports are used only for reading and some are used both for reading and writing: let us denote their number as $\mathbf{r}_k$ and $\mathbf{rw}_k$, respectively. Therefore, we have

$$\mathbf{x}_k = \mathbf{r}_k + \mathbf{rw}_k. \tag{20}$$

Each CU has a different amount of data to read through the AXI interface, so the data read time also varies from kernel to kernel. At the beginning, all the ports share the bandwidth, but when the first CU finishes reading, the available bandwidth for the remaining CUs increases, since the number of active reading ports is reduced. Eventually there will be only one active port reading data from external DDR memory.

Worst-case approximation: Unfortunately, taking into account the different read times requires an iterative formulation, which would be too costly to implement (the MINLP solver already times out with just a single iteration). Therefore, we simplify it to obtain a worst-case formula by assuming that the number of active AXI ports is always equal to the initial value, i.e., $NR_f = \sum_{k\in\mathbf{K}}(\mathbf{x}_k \cdot n_{k,f})$, i.e. that the memory reading times are roughly balanced among the kernels. In this way, we can use a fixed value for the read bandwidth as in (19), and consequently the *approximated reading time* becomes

$$TR_{k,f} \simeq \frac{dr_k}{BR_{k,f}}. \tag{21}$$

2) Writing to local DDR: All the CUs of kernel k write their output data dw_k to the local DDR memory roughly at the same time, while the CUs of different kernels in principle can write at different times. Since it is difficult to model the exact time at which each kernel starts writing the data, we consider the worst-case scenario when all the CUs of all kernels start writing at the same time, in the same way as we did for the reading phase. This is a crude approximation, but we consider it acceptable because the writes are much fewer than the reads. Therefore, we will determine the $TW_{k,f}$ time in a similar way as we obtained the $TR_{k,f}$ time. One difference is that there is no output data duplication:

$$dw_k = \frac{\mathbf{DO}_k}{N_k}. \tag{22}$$

Each CU of kernel k writes in the local DDR through $\mathbf{y}_k$ separate AXI ports, each with bandwidth BX_f. Some of these ports are used both for writing and reading ($\mathbf{rw}_k$), while some only for writing ($\mathbf{w}_k$). Hence, we have

$$\mathbf{y}_k = \mathbf{w}_k + \mathbf{rw}_k. \tag{23}$$

The DDR memory write bandwidth is $\mathbf{BDW}$. It is instantaneously shared among the NW_f *actively writing* AXI ports associated to the various kernels allocated to that FPGA. NW_f changes over time and we assume that initially it takes the value $NW_f = \sum_{k\in\mathbf{K}}(\mathbf{y}_k \cdot n_{k,f})$. The instantaneous write bandwidth for each CU is therefore

$$BW_{k,f} = \mathbf{y}_k \cdot \min\left(BX_f, \frac{\mathbf{BDW}}{NW_f}\right). \tag{24}$$

Worst-case approximation: As in the previous case, we can obtain a worst-case expression by assuming that the number of active AXI ports is always equal to the initial value, i.e., $NW_f = \sum_{k\in\mathbf{K}}(\mathbf{y}_k \cdot n_{k,f})$. By assuming that the write bandwidth is always as in (24), we obtain the *approximated writing time*:

$$TW_{k,f} \simeq \frac{dw_k}{BW_{k,f}}. \tag{25}$$

3) Computing: Let us define $\mathbf{TC1}_k$ as the worst case computing time when kernel k is implemented with only **one CU** and runs at clock frequency $\mathbf{F1}_k$. The computing latency in clock cycles needed by one CU is therefore $\mathbf{L1}_k = \mathbf{TC1}_k \cdot \mathbf{F1}_k$. Considering that kernel k is arbitrarily parallelizable, its latency L_k scales proportionally to its number of CUs, N_k:

$$L_k = \frac{\mathbf{L1}_k}{N_k}. \tag{26}$$

The actual clock frequency in each FPGA depends on both resource utilization and the different kernels allocated to it. We observed an almost linear graceful degradation of clock frequency for each kernel as the amount of resources increases:

$$F_{k,f} = \mathbf{F1}_k - \psi \cdot R_f, \tag{27}$$

where R_f is a metric of resource utilization in the FPGA f and $\psi \geq 0$ is a constant, potentially different for each kernel. To obtain ψ, we collected experimental data with different numbers of compute units (in this case, the kernel resource utilization will change) and we noticed that a linear fitting worked very well. Since all kernels in f run at the same clock frequency[6], F_f, it is determined as

$$F_f = \min_k F_{k,f} \tag{28}$$

and we can obtain the computing delay for each kernel k in FPGA f:

$$TC_{k,f} = \frac{L_k}{F_f}. \tag{29}$$

IV. Geometric Programming and Allocator

The optimization problem discussed in the previous section can be solved by a Mixed-Integer Non-Linear Programming (MINLP) solver like Couenne [3]. This can lead, however, to impractically long optimization times for designs with many kernels and FPGAs, as we will show in the next section. Consider, for instance, that the VGG-net convolutional neural network with 20 layers spread on eight FPGAs has 160 integer variables. Using a slow MINLP solver within a design space exploration loop often leads to prohibitively long run times.

For this reason, we propose a heuristic formulation that separates the optimization in two steps. The first step determines

[6]Even though it would be possible for each kernel to run at a different clock frequency even in the same FPGA, we did not consider this possibility for now.

the total *fractional* number of CUs for each kernel to minimize the computation time (this simplification is reasonable when the reading time and writing time are much smaller than the computation time, which is the case for CNNs). With this relaxation, we can use a Geometric Programming (GP) solver that is much faster than a MINLP solver (just like Linear Programming is much faster than its integer variant). The second step allocates the CUs to the available FPGAs in a greedy but "smart" way, in order to minimize the data transfer time between the host CPU and the external DDR memory. In the following, we refer to this two-step approach as **GP+A**.

A. Geometric Programming

To use GP [19], we relax the constraints of the problem by allowing the total number of CUs for each kernel $n_{k,f}$ be a real number, rather than an integer as it should be. Given the number of FPGAs, the total available resources, and the computation time of each kernel (since only the computation time depends on resources like DSPs and BRAM), a GP solver returns the optimal number of CUs of each kernel as real numbers. With these, the allocation problem becomes fully symmetric across the F identical FPGAs, and the optimum solution has an equal distribution of CUs across the F FPGAs.

Let us define $\widehat{n}_k \in \mathbb{R}$ the number of CUs that would be assigned to an FPGA. The total number of CUs of kernel k will be

$$\widehat{N}_k = \mathsf{F} \cdot \widehat{n}_k. \tag{30}$$

To guarantee that at least one CU is generated for each kernel, we need to specify that $\widehat{N}_k \geq 1$, but of course it is possible that $\widehat{n}_k \leq 1$[7].

Now the kernel latency becomes

$$\widehat{L}_k = \frac{\mathsf{L1}_k}{\widehat{N}_k}, \ \forall k \in K \tag{31}$$

and the kernel computing time becomes

$$\widehat{TC}_k = \frac{\widehat{L}_k}{\mathsf{F1}_k}, \ \forall k \in K, \tag{32}$$

where we use $\mathsf{F1}_k$ as an estimation of the actual clock frequency. This is justified by the fact that the clock frequencies of different kernels are similar, as we will show in Section V, and that the degradation due to the implementation affects all FPGAs in a similar way, since we utilize them fairly uniformly.

Thus, we can reformulate the optimization problem in (2)–(4) as follows:

$$\text{minimize} \quad \widehat{TC} \tag{33}$$

$$\text{subject to}$$

$$\widehat{TC} \geq \widehat{TC}_k, \quad \forall k \in K \tag{34}$$

$$\widehat{N}_k \geq 1, \quad \forall k \in K \tag{35}$$

$$\textstyle\sum_k \frac{\widehat{N}_k}{\mathsf{F}} \mathsf{R}_{k,t} \leq \mathsf{R}_t. \tag{36}$$

The new formulation in (33)–(36) is compatible with GP requirements [19], and as such can be solved very efficiently.

[7]We can liken $\widehat{n}_k$ to the *average* number of CUs of kernel k across F FPGAs.

Once we obtain the (fractional) number of CUs of each kernel, in the next step we allocate them on FPGAs in integer chunks, via discretization. Note that the initial GP spreads the kernels across F, which is clearly suboptimal because it increases the data transfer time and the complexity of the work done by the host CPU. This is why we introduce a heuristic allocator to optimize the mapping.

B. FPGA allocation

Before allocation, the variables $\widehat{N}_k \in \mathbb{R}$ must be discretized to obtain $N_k \in \mathbb{N}$. We enforce integrality using a branch-and-bound technique similar to those used in Integer Linear Programming. We generate two sub-problems, each with $N_k \leq \lfloor \widehat{N}_k \rfloor$ and $N_k \geq \lceil \widehat{N}_k \rceil$. The search is pruned when the overall resource usage of a sub-problem exceeds the resource bound of all the FPGAs (this might happen because GP uses $\widehat{N}_k$ to meet the resource constraints, but $\lceil \widehat{N}_k \rceil \geq \widehat{N}_k$). Even though this branch-and-bound technique may lead to a worst-case exponential branching tree, in practice this does not lead to excessive execution times due to:

- the pruning strategy,
- the fact that we need to discretize only K variables, where K is the total number of of distinct kernels in the network, and
- the fact that the number of kernels K is relatively small. E.g., it is around 20 for the VGG benchmark, and 37 for the ResNet benchmark. ResNet, however, includes only 16 types of distinct kernels, and different layers with the same type of kernel can have exactly the same number of total CUs. Hence even for ResNet we have only 16 variables to discretize, as discussed below.

The full MINLP approach, on the other hand, must discretize every variable (160 in the case of VGG over eight FPGAs), hence it may potentially have a much larger branching tree.

For each sub-problem generated with the discretization, we perform the actual allocation, which consists of two phases:

1) Kernel group allocation.
2) Individual kernels allocation.

1) Kernel group allocation: To minimize the H2F and F2H transfer times, we try to allocate on the same FPGA kernels that are consecutive in the pipeline, so that their communication can happen through buffers in local DDR without involving the host CPU. To do so, we first enumerate all possible groups of at least two kernels. We then associate each group with the size of the input data required by the kernels in the group, data that will be transferred locally if the group fits in a single FPGA. An example of groups and associated data is shown on the left of Fig. 7(a).

Many of these combinations are not feasible (i.e., the group cannot fit in one FPGA) and are therefore flagged as invalid and pruned, as shown in the figure. This is beneficial because it reduces the overall runtime of our heuristic. After pruning, we sort the list of remaining kernel groups in descending order of input data size, as shown on the right of Fig. 7(a).

Based on this list and on FPGA resource constraints, we allocate the groups using the greedy heuristic procedure called *AllocateGroups* in Algorithm 1.

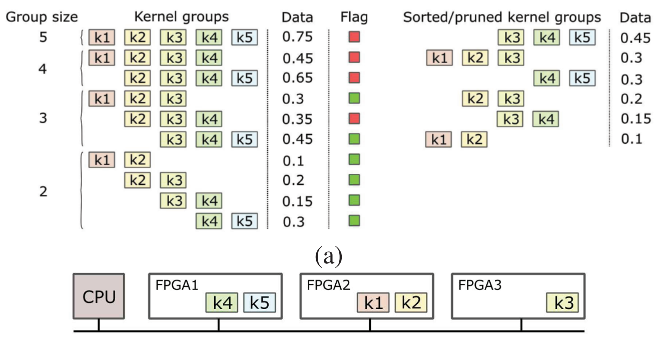


Fig. 7: Grouping example with five kernels: (a) possible kernel groups (left), flagging and discarding, and kernel group sorting by input data size (right). (b) possible allocation: first allocate the kernel groups and then allocate the individual kernels.

Algorithm 1: Pseudo-code of kernel group allocation

```
1  procedure AllocateGroups($N_k$, $R$)
2      $\mathrm{CU} = (CU_1, CU_2, \ldots, CU_{\mathsf{K}})$   // Kernel CUs to allocate
3      $\mathrm{CU}_k = N_k, \forall k$                 // Initialized to GP values
4      $S = (S_1, S_2, \ldots, S_{\mathsf{F}})$            // FPGA resource slacks
5      $S_f = R, \forall f$      // Initialized to resource constraint
6      $n_{k,f} = 0, \forall k, f$   // Allocated CUs initialized to zero
7      $N_g$ = All_Kernel_groups // Set of all possible groups
8      $N_g = N_g \setminus$ Infeasible_groups($N_g$)             // Pruning
9      $N_g$ = Sorted_groups($N_g$)          // Sorting by data size
10     $R_g$ = Group_resources($N_g$)         // Resources needed by
                                          // each group
11     for $n = 1$ to $|N\_g|$ do           // Try to allocate group $n$
12
13         for $f = 1$ to $\mathsf{F}$ do
14             if $R\_g[n] \leq S_f$ then         // if $f$ has space left
15
16                 $S_f = S_f - R\_g[n]$
17                 for $k \in N\_g[n]$ do         // all kernels in $n$
18
19                     $n_{k,f} = \mathrm{CU}_k$
20                     $\mathrm{CU}_k = 0$
21                 $N_g = N_g \setminus \{N_g[n]\}$         // remove group $n$
22                 sortFPGA($S$)     // Sort by increasing slack
23
24     return CU, $S$
```

After pruning and sorting the groups (N_g is the set of groups and its cardinality is $|N\_g| = \sum_{n=1}^{\mathsf{K}-1} n$), the loops in lines 11–23 simply try to allocate each group as long as an FPGA has enough space. If a group is allocated to FPGA f, each kernel k in the group will have all of its CUs (as determined by GP) allocated to f ($n_{k,f} = \mathrm{CU}_k = N_k$) and the corresponding value CU_k will be set to zero, otherwise CU_k will keep the initial value N_k. The resource slack of f is also updated. The procedure returns the modified arrays CU and S, which are then passed to the last phase for the individual allocation of the residual kernels. Fig. 7(b) shows one possible allocation of a five-kernel application. By following the order of the sorted kernel groups, the allocator first tries to allocate the first two kernel groups on a single FPGA, but does not succeed. Then it tries to allocate the third kernel group and successfully assigns it to FPGA1. Similarly, k1 and k2 are allocated on FPGA2. The individual kernel k3 cannot fit on FPGA1 or FPGA2, and is allocated on FPGA3 using the algorithm in Section IV-B2.

2) Individual kernel allocation: Before delving into the details of the procedure shown in Algorithm 2, it is important to note that, due to the discretization that follows the GP solution, it might happen that an allocation is not feasible, as it might exceed the initial resource constraint $R_c = R$. For this reason, we use a *soft* bound that can be increased iteratively by a little amount ($R_c = R_c + \Delta$) until it exceeds the initial constraint by a predetermined threshold ($R_c > R + T$). This is implemented by the outer **while** loop in lines 4–37 of Algorithm 2, with the boundary increased on line 35 and the exit condition (in case of allocation) on line 37.

Algorithm 2: Pseudo-code of kernel allocation

```
1  procedure AllocateKernels(CU, $S$)
2      $R_c = R$              // Resource constraint initialized
3      alloc = FALSE
4      while $R_c < R + T$ and not alloc do
5          sortKernels(CU)   // Sort by descending criticality
6          for $k = 1$ to $\mathsf{K}$ do // Allocate large kernels first
7              $f = 1$
8              while $\mathrm{CU}_k \cdot R_k > R$ do // Can't fit in one FPGA
9                  if $S_f = R$ then
10                     $\delta\mathrm{CU} = \lfloor R/R_k \rfloor$
11                     $\mathrm{CU}_k = \mathrm{CU}_k - \delta\mathrm{CU}$
12                     $S_f = S_f - \delta\mathrm{CU} \cdot R_k$
13                     $n_{k,f} = n_{k,f} + \delta\mathrm{CU}$
14                 else
15                     $f = f + 1$

16         sortFPGA($S$)                // Sort by ascending slack
17         for $k = 1$ to $\mathsf{K}$ do // Allocate all kernels
18             partial_alloc = FALSE
19             $f = 1$
20             while $f \leq \mathsf{F}$ and not partial_alloc do
21                 if $S_f \geq \mathrm{CU}_k \cdot R_k$ then
22                     $S_f = S_f - \mathrm{CU}_k \cdot R_k$
23                     $n_{k,f} = n_{k,f} + \mathrm{CU}_k$
24                     $\mathrm{CU}_k = 0$
25                     partial_alloc = TRUE
26                 $f = f + 1$
27             if not partial_alloc then
                   // Use least used FPGA F, if possible
28                 $\delta\mathrm{CU} = \lfloor S_{\mathsf{F}}/R_k \rfloor$
29                 $\mathrm{CU}_k = \mathrm{CU}_k - \delta\mathrm{CU}$
30                 $S_{\mathsf{F}} = S_{\mathsf{F}} - \delta\mathrm{CU} \cdot R_k$
31                 $n_{k,\mathsf{F}} = n_{k,\mathsf{F}} + \delta\mathrm{CU}$
32             sortFPGA($S$)
33         if $\sum_k \mathrm{CU}_k > 0$ then // Not all CUs are allocated
34             $R_c = R_c + \Delta$
35         else
36             alloc = TRUE             // All kernels allocated
37     if alloc then // All CUs are allocated
38         return $n_{k,f}, \forall k, \forall f$
39     else
40         return allocation failed
```

If no discretization case can be allocated (*alloc* = FALSE for all of them), it means that the initial constraint R was too tight and the entire GP+A heuristic needs to be run again with the looser constraint $R + T_{\max}$.

The two **for** loops inside the **while** loop (lines 6–15 and 18–37, respectively) are preceded by a procedure that sorts the kernels in descending "criticality." Critical kernels are those that might end up being the slowest in the pipeline and determine the overall II. In practice, we sort the kernels in descending $\widehat{TC}_k$ as determined by the GP step.

After sorting by criticality, the first **for** loop attempts to allocate a portion of the CUs of the large kernels that cannot fit in a single FPGA (line 8) to still *empty* FPGAs (line 9).

The second loop is preceded by an FPGA sorting by *ascending slack* (the less empty first). The rationale is that we want to consolidate the kernels by allocating *all* the residual CUs to the already partially filled FPGAs. If this is not possible (line 28), we use the least used FPGA, which is the last in the ordered set (**F**, i.e., the one with the largest slack), to allocate as many CUs as possible.

Before the next iteration of the **for** loop, the FPGAs are sorted again by ascending slack.

After the loop, if there are still CUs that are not allocated (line 33), the soft boundary is increased and the outer **while** loop is executed again.

If all kernels are allocated, the procedure returns $n_{k,f}$ for all kernels and FPGAs. In this case, the FPGA working frequency is updated and the AXI reading time T_{read} and writing time T_{write} are calculated, as well as the data transfer time between the host CPU and the local DDR memory T_{h2f} and T_{f2h}. Finally, the II is computed and compared with the best obtained so far. If better, the allocation of the current sub-problem obtained with discretization of GP results is kept, otherwise it is discarded and a new discretization is considered.

V. Experimental results

We implemented our allocation heuristics in C++ and linked it to an existing GP solver [20]. To validate our optimization method, we use several widely used CNNs: AlexNet [6], VGG-net [7], YOLO [4] and ResNet [8]. For AlexNet, we consider both a 32-bit floating point version and a 16-bit fixed-point version, which we denote Alex-32 and Alex-16, respectively. For VGG-net, we only use the 16-bit fixed-point version, denoted VGG-16. For YOLO we only use the floating-point version, denoted YOLO-32. Finally, for ResNet we only use 16-bit fixed-point version denoted RESNET-16. Again, this is just an arbitrary selection of benchmarks to show the effectiveness of our technique for a growing CNN complexity. We validate our heuristic against the MINLP solver Couenne under the same conditions, and for this purpose we introduce two symbols:

- **GP+A** refers to the solution given by the heuristic method that couples **GP** and **Allocation**;
- **MINLP** refers to the solution obtained using the state-of-the-art **MINLP** solver Couenne.

We compare the solutions obtained with the two methods for different numbers of FPGAs and different resource constraints. We ran all our MINLP and GP+A optimizations on a multi-core CPU (Intel Core i7-6900K clocked at 3.2 GHz, 16 cores) with Linux CentOS 6.9, and our hardware experiments on an AWS F1.x16large instance with eight UltraScale Plus FPGAs.

Initially, we ran our kernels individually on AWS and obtained the performance and cost characteristics with one CU each, that are needed for the cost-performance model. Tables III–V report the input/output data size of each kernel, duplication factor of the input data, constant data weights, number of input/output data ports, working frequency, resource usage (since the critical resource usage in our applications are DSPs, we only report the DSP usage), and computation

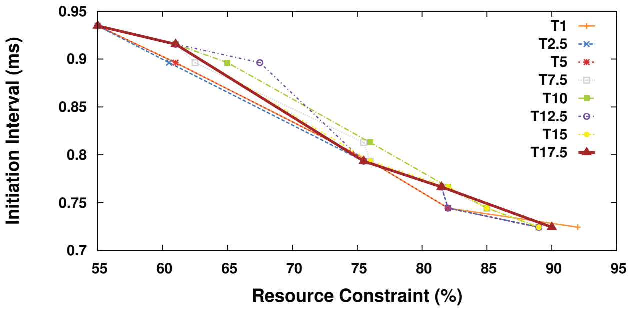


Fig. 8: II vs. R_{max} with different resource usage thresholds for AlexNet fixed-point (Alex-16) on two FPGAs.

time[8]. Note that we do not need to characterise all kernels individually, because some of them have exactly the same configuration (same input/output data size and amount of computation).

Out of all the experiments that we carried out, we select five representative cases of increasing complexity: ALEX-16 on two FPGAs, ALEX-32 on four FPGAs, YOLO-32 on three FPGAs, VGG-16 on four FPGAs, VGG-16 on six FPGAs, and ResNet on five FPGAs. The MINLP solver manages to complete and return the (provably) optimum solution in a reasonable time only in the smallest among all these cases, namely ALEX-16 on two FPGAs. The MINLP CPU time for this case is shown in Table VI, where we vary the DSP resource constraint (FPGA DSPs are always the limiting factor) from 55% to 92%. In this range, we observe an almost linear degradation of the maximum clock frequency with the FPGA resource utilization, which we captured in (27).

For all the other cases, we had to set a time limit to stop the MINLP solver. We chose it by looking at the progress of the solution: when we observed a flattening of the II curve as in Fig. 3, we decided to stop the solver. The time limit, as shown in Table VII, varies from 10 to 70 hours for the different cases, due to the different size of the problem.

Table VIII shows instead the CPU time required by our heuristic to generate a set of results, which is generally several thousand times faster than the MINLP solver.

As shown in Algorithm 2, our heuristic requires to set a resource usage threshold, T. Fig. 8 shows the effect of changing it while keeping the other parameters of ALEX-16 on two FPGAs constant. We observe little effect of T on the value of II across a large range of the resource constraint R. Similar results are obtained for the other benchmark cases. Because of this, in the following we report the results obtained with one specific threshold, namely for $T = 1\%$.

The plots in Fig. 9 show the results obtained by changing the resource constraint for both the MINLP solver and our heuristic. ALEX-16 on two FPGAs, shown in Fig. 9(a), shows the effectiveness of our method since we know that MINLP returns the optimum result for this benchmark: notice how MINLP and GP+A completely overlap. Interestingly, for all the other cases with increased complexity shown in Figs. 9(b)–(e), GP+A significantly outperforms MINLP (even for runs

[8]To save space, we did not include the ResNet characterization table.

TABLE III: Characterization of kernels for Alex-32 (floating point) and Alex-16 (fixed-point). C, P, N stand for convolutional, pooling and normalization layers.

	Alex-32									Alex-16								
Kernels	$\mathsf{DI_k}$ (MB)	$\mathsf{DO_k}$ (MB)	$\mathsf{C_k}$ (MB)	δ_k	γ_k	$\mathsf{rw_k}$	$\mathsf{F1_k}$ (GHz)	DSP (%)	$\mathsf{TC1_k}$ (ms)	$\mathsf{DI_k}$ (MB)	$\mathsf{DO_k}$ (MB)	$\mathsf{C_k}$ (MB)	δ_k	γ_k	$\mathsf{rw_k}$	$\mathsf{F1_k}$ (GHz)	DSP (%)	$\mathsf{TC1_k}$ (ms)
C1	0.62	1	0	0	1	1	0.25	21.24	4.41	0.31	0.58	0	0	1	1	0.25	4.31	2.63
P1	1	0.27	0	1	1	1	0.25	0	0.11	0.58	0.139	0	1	1	1	0.25	0.58	0.37
N1	0.27	0.27	0	1	1	1	0.25	2.11	0.29	0.139	0.139	0	1	1	1	0.25	0.06	0.28
C2	0.27	0.17	1.17	0	1	1	0.22	37.59	2.99	0.139	0.086	0.614	0	1	1	0.25	7.63	1.927
N2	0.17	0.17	0	1	1	1	0.223	7.75	0.2	0.086	0.086	0	1	1	1	0.25	0.06	0.17
C3	0.17	0.25	3.375	0	1	1	0.214	28.13	2.18	0.086	0.13	1.77	0	1	1	0.25	5.66	1.82
C4	0.25	0.25	2.53	0	1	1	0.21	37.5	1.82	0.13	0.13	1.33	0	1	1	0.25	7.55	1.08
C5	0.25	0.035	1.69	0	1	1	0.22	37.5	3.73	0.13	0.018	0.884	0	1	1	0.25	7.55	1.72

TABLE IV: Characterization of kernels (K) for YOLO-32 (floating point). C and P stand for convolutional and pooling layers.

K	$\mathsf{DI_k}$ (MB)	$\mathsf{DO_k}$ (MB)	$\mathsf{C_k}$ (MB)	δ_k	γ_k	$\mathsf{rw_k}$	$\mathsf{F1_k}$ (GHz)	DSP (%)	$\mathsf{TC1_k}$ (ms)
C1	0.574	3.063	0.0016	1	0	1	0.25	3.66	6.63
P1	3.063	0.767	0	1	1	1	0.25	0	0.43
C2	0.767	1.531	0.018	1	0	1	0.25	9.52	4.22
P2	1.531	0.383	0	1	1	1	0.25	0	0.03
C3	0.383	0.766	0.07	0	1	1	0.25	9.43	2.24
P3	0.766	0.191	0	1	1	1	0.25	0	0.03
C4	0.191	0.383	0.281	0	1	1	0.25	18.77	1.2
P4	0.383	0.096	0	1	1	1	0.25	0	0.03
C5	0.096	0.096	0.563	0	1	1	0.25	18.72	0.58
P5	0.096	0.024	0	1	1	1	0.25	0	0.016
C6	0.024	0.048	1.125	0	1	1	0.247	4.68	1.02
C7	0.048	0.079	0.415	0	1	1	0.25	7.31	0.49

TABLE V: Characterization of kernels (K) for VGG-16 (fixed-point). C and P stand for convolutional and pooling layers.

K	$\mathsf{DI_k}$ (MB)	$\mathsf{DO_k}$ (MB)	$\mathsf{C_k}$ (MB)	δ_k	γ_k	$\mathsf{rw_k}$	$\mathsf{F1_k}$ (GHz)	DSP (%)	$\mathsf{TC1_k}$ (ms)
C1	0.287	6.126	0.003	1	0	1	0.25	2.95	14.652
C2	6.126	6.126	0.07	1	0	1	0.249	15.14	20.18
P2	6.126	1.531	0	1	0	1	0.25	0.03	0.115
C3	1.531	3.063	0.141	1	0	1	0.25	15.14	10.042
C4	3.063	3.063	0.281	1	0	1	0.249	15.14	13.71
P4	3.063	0.766	0	1	1	1	0.25	0.03	0.115
C5	0.766	1.531	0.563	0	1	1	0.246	15.07	7.808
C6	1.531	1.531	1.125	0	1	1	0.249	15.05	14.97
C7	1.531	1.531	1.125	0	1	1	0.249	15.05	14.97
P7	1.531	0.383	0	1	1	1	0.25	0.03	0.115
C8	0.383	0.766	2.25	0	1	1	0.244	15.02	7.66
C9	0.766	0.766	4.5	0	1	1	0.25	15.02	14.94
C10	0.766	0.766	4.5	0	1	1	0.25	15.02	14.94
P10	0.766	0.192	0	1	1	1	0.25	0.01	0.115
C11	0.192	0.192	4.5	0	1	1	0.245	14.99	3.84
C12	0.192	0.192	4.5	0	1	1	0.245	14.99	3.84
C13	0.192	0.192	4.5	0	1	1	0.245	14.99	3.84

TABLE VI: ALEX-16 on 2 FPGAs: MINLP CPU time to obtain one optimum solution varying the resource constraint.

	Resource Usage on each FPGA				
	55%	**61%**	**76%**	**82%**	**92%**
Time (h)	8.2	1.7	1.8	1.93	1.6

TABLE VII: Time limit used by the Couenne MINLP solver to obtain one point on the II vs. R curve of each implementation in Fig. 9.

	CNN / # FPGAs			
	Alex-32 4 FPGAs	**YOLO-32 3 FPGAs**	**VGG-16 4 FPGAs**	**VGG-16 6 FPGAs**
Time (h)	10	30	30	40

TABLE VIII: Execution time of our heuristic method GP+A to generate the Pareto points in Fig. 9.

	CNN / # FPGAs				
	Alex-16 2 FPGAs	**Alex-32 4 FPGAs**	**YOLO-32 3 FPGAs**	**VGG-16 4 FPGAs**	**VGG-16 6 FPGAs**
Time (s)	25	22	17	89	66

within the fairly large time limits of Table VII), with only one exception: the point at $R = 61\%$ for ALEX-32 on four FPGAs in Fig. 9(b) where the heuristic is slightly worse than MINLP.

Different from the other benchmarks, for ResNet the comparison between the heuristic and the MINLP solver is impractical. In the 5-FPGA case for which we report the heuristic result in Fig. 10, the MINLP solver could not return a feasible solution even after a very long runtime. We stopped it after 70 hours, whereas our heuristics returned the points in Fig. 10 in only 15 seconds.

In general, the larger the size of the problem, the larger the gap between GP+A and MINLP. As the problem gets more complex, the MINLP solver either gets stuck in a local minimum, or needs an impractical amount of time to converge to the global optimum. Our heuristic instead returns in a short amount of time a competitive solution.

The histograms in Fig. 11 and Fig. 12 show the resource allocation of kernels for ALEX-32 on four FPGAs and VGG-16 on six FPGAs, respectively, with a different value of R. These correspond to two specific points that are circled out in the plots of II vs R in Fig. 9(b) and Fig. 9(e), respectively.

Fig. 11 shows that MINLP and GP+A made very similar allocations. Both manage to place in the same FPGA kernels that are consecutive in the pipeline, as highlighted by the coloring (similar colors refer to consecutive kernels that should be allocated on the same FPGA).

On the contrary, in the more complex case in Fig. 12 the allocations are significantly different. While GP+A manages

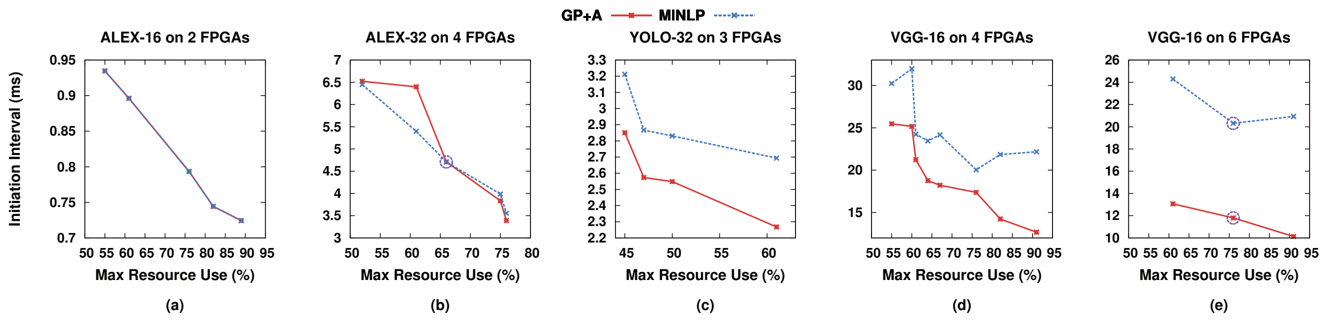


Fig. 9: Initiation interval as a function of FPGA resource usage: (a) ALEX-16 on 2 FPGAs, (b) ALEX-32 on 4 FPGAs, (c) YOLO-32 on 3 FPGAs, (d) VGG-16 on 4 FPGAs and (e) VGG-16 on 6 FPGAs.

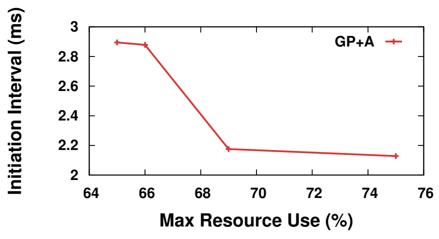


Fig. 10: RESNET-16 on 5 FPGAs: II vs resource usage.

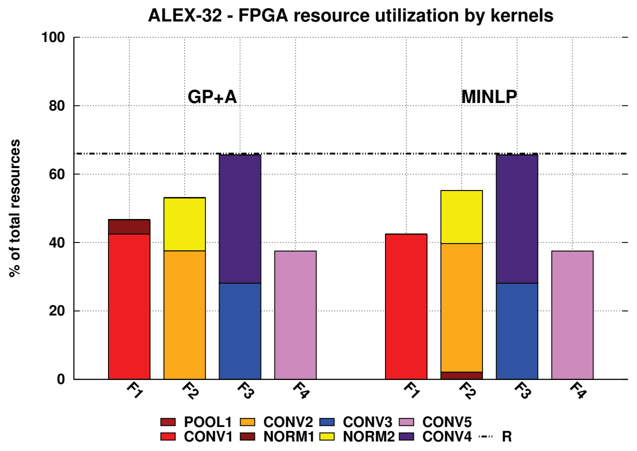


Fig. 11: ALEX-32 allocation on 4 FPGA using GP+A and MINLP.

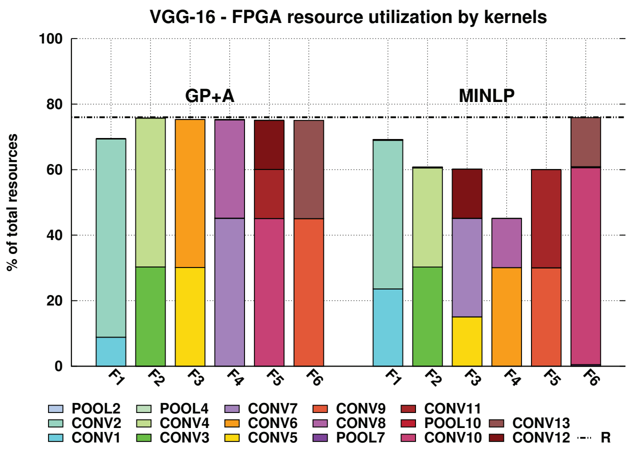


Fig. 12: VGG-16 kernel allocation on 6 FPGA using GP+A and MINLP.

to both use efficiently the resources available within the R constraint and group in the same FPGA consecutive kernels, MINLP does not succeed at any of these two tasks within the allotted time.

In Fig. 13, we show the value of II (red curves) as a function of the number of FPGAs, for the best solutions returned for each number of FPGA by our heuristic in the four benchmark cases. We plot in the same graphs the Transfer time and Computing time fractions of II, which show that there is an optimum number of FPGAs for each application. This is because more FPGAs (1) provide more parallel resources that allow decreasing the computing time, but (2) more FPGAs also tend to increase the transfer time in the H2F and F2H phases because fewer kernel pairs can share data directly and data transfers via host code are slower. Even though the MINLP solver can theoretically return this optimum, for the more complex cases this is highly impractical. Our heuristic can be efficiently used for a faster design space exploration.

VI. Conclusion

We have proposed and experimentally analyzed a fast and effective method to allocate resources for each kernel in a multi-kernel task-level pipelined application, like a CNN, to optimize the throughout on multiple FPGAs. Our heuristic optimizes the number of compute units of each kernel and their allocations, while respecting resource constraints and taking into account the cost of data transfer times between the FPGAs and a host CPU. We developed a cost/performance model, we modeled it as an optimization problem, and we solved it using a MINLP solver. However, due to the long CPU time and inefficiency of the solver, we propose a fast and accurate heuristic method that consists of two main parts. First we use a GP solver (using a relaxed representation of the same model, without integrality constraints) to get the number of CUs. Then we use a heuristic allocator to assign them to different FPGAs in order to minimize the data transfer time. Experimental results show that our heuristic method can provide very similar results as the exact MINLP solution when

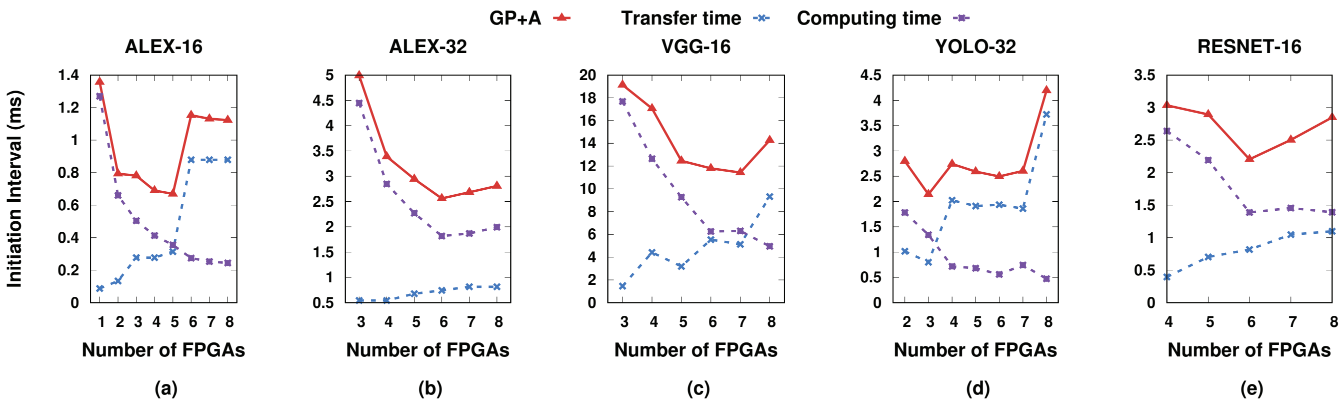


Fig. 13: Initiation interval as a function of the number of FPGAs used.

the problem size is small, and it returns much better results for larger problem sizes.

Future work can cover power and energy consumption, consider streaming, and improve the kernel-to-kernel communication model. It would also be interesting to extend the method to work in a hierarchical fashion, where the performance of each kernel (as well as its cost, bandwidth requirements and so on) depends not only on the number of CUs at the top level, but also on the unrolling of loops inside each kernel.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[2] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060

[3] P. Belotti. (2018) Couenne (convex over and under envelopes for nonline estimation). [Online]. Available: https://www.coin-or.org/Couenne/

[4] J. Pedoeem and R. Huang, "YOLO-LITE: A real-time object detection algorithm optimized for non-gpu computers," *CoRR*, vol. abs/1811.05588, 2018. [Online]. Available: http://arxiv.org/abs/1811.05588

[5] J. Shan, M. R. Casu, J. Cortadella, L. Lavagno, and M. T. Lazarescu, "Exact and heuristic allocation of multi-kernel applications to multi-fpga platforms," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: ACM, 2019, pp. 3:1–3:6.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016. [Online]. Available: http://dx.doi.org/10.1109/CVPR.2016.90

[9] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 151–162, Oct. 2006.

[10] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 535–547.

[11] K. Guo, L. Sui, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto customized hardware," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2016, pp. 24–29.

[12] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: An automated tool for building high-performance dnn hardware accelerators for fpgas," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: ACM, 2018, pp. 56:1–56:8.

[13] W. Jiang, E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Achieving super-linear speedup across multi-fpga for real-time dnn inference," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, p. 1–23, Oct 2019.

[14] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ser. ISLPED '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 326–331. [Online]. Available: https://doi.org/10.1145/2934583.2934644

[15] W. Zhang, J. Zhang, M. Shen, G. Luo, and N. Xiao, "An efficient mapping approach to large-scale dnns on multi-fpga architectures," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 1241–1244.

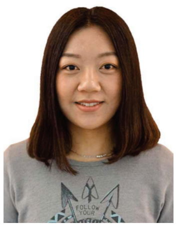
[16] W. Jiang, E. H. Sha, Q. Zhuge, L. Yang, X. Chen, and J. Hu, "Heterogeneous fpga-based cost-optimal design for timing-constrained cnns," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2542–2554, Nov 2018.

[17] J. Shen, D. Wang, Y. Huang, M. Wen, and C. Zhang, "Scale-out acceleration for 3d cnn-based lung nodule segmentation on a multi-fpga system," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, June 2019, pp. 1–6.

[18] J. Shen, D. Wang, Y. Huang, M. Wen, and C. Zhang, "Accelerating 3d cnn-based lung nodule segmentation on a multi-fpga system," in *FPGA*, 2019.

[19] S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi, "A tutorial on geometric programming," *Optimization and engineering*, vol. 8, no. 1, p. 67, 2007.

[20] E. Burnell and W. Hoburg. (2018) Gpkit. [Online]. Available: https://github.com/convexopt/gpkit



Junnan Shan received the B.S. and M.S. degrees from Politecnico di Torino, Italy, where she is currently pursuing the Ph.D. degree with the Department of Electronics and Telecommunications under the supervision of Prof. Mario Casu and Prof. Luciano Lavagno. Her research interests focus on electronic design automation, system-level design, low-power, and high-performance computing, and high-level synthesis.

Mihai Teodor Lazarescu received the Ph.D. degree in Electronics and Communications from Politecnico di Torino (Italy) in 1998, where he serves now as Assistant Professor. He was Senior Engineer at Cadence Design Systems and founded several startups. He co-authored more than 60 scientific publications, four books, and international patents. His research interests include design tools for reusable WSN platforms, sensing, indoor localization, and data processing for IoT, low power embedded design, high-level HW/SW co-design, and high-level synthesis.

Jordi Cortadella (S'87-M'89-F'15) is a Professor with the Computer Science Department, Universitat Politècnica de Catalunya, Barcelona, Spain. His current research interests include formal methods and computer-aided design of VLSI systems with a special emphasis on asynchronous circuits, concurrent systems, and logic synthesis. Prof. Cortadella is a member of Academia Europaea. He received best paper awards at the International Symposium on Advanced Research in Asynchronous Circuits and Systems in 2004 and 2016, the Design Automation Conference in 2004, and the International Conference on Application of Concurrency to System Design in 2009. He has served on the technical committees of several international conferences in the field of design automation and concurrent systems, and is an Associate Editor of the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Luciano Lavagno received the Ph.D. degree in EECS from the University of California at Berkeley, Berkeley, CA, USA, in 1992. He was the architect of the POLIS HW/SW co-design tool. From 2003 to 2014, he was an Architect of the Cadence CtoSilicon high-level synthesis tool. Since 1993, he has been a Professor with the Politecnico di Torino, Italy. He co-authored four books and more than 200 scientific papers. His research interests include synthesis of asynchronous circuits, HW/SW co-design, high-level synthesis, and design tools for wireless sensor networks.

Mario R. Casu (M'04-SM'18) received the Ph.D. degree in electronics and communications engineering from the Politecnico di Torino, Torino, Italy, in 2001, where he is currently an associate professor. His research interests are Systems-on-Chip with specialized accelerators, System-level design and design methodology for FPGAs and ASICs, and Embedded Machine Learning. He is also interested in the design of circuits, systems, and platforms for industrial applications (biomedical, automotive, food). His past work focused mostly on latency-insensitive design of Systems-on-Chip (SoC) and on Networks-on-Chip. He regularly serves in the technical program committee of international conferences (among which DAC, ICCAD, DATE).