ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electrical, Electronics and Communications Engineering
(33.rd cycle)

Performance and Power Optimization of Multi-kernel Applications on Multi-FPGA Platforms

Junnan Shan
* * * * * *

Supervisors
Prof. Mario Roberto Casu
Prof. Luciano Lavagno

Doctoral Examination Committee:
Prof. Dirk Koch, Referee, University of Manchester
Prof. Davide Quaglia, Referee, University of Verona
Prof. William Fornaciari, Polytechnic of Milan
Prof. Roberto Passerone, University of Trento
Prof. Mihai Teodor Lazarescu, Polytechnic of Turin

Politecnico di Torino
March, 2021

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . .
Junnan Shan
Turin, March, 2021

Summary

Applications using Convolutional Neural Networks (CNNs) and other Deep Neural Networks (DNNs) for Machine Vision and Natural Language Processing tasks achieved breakthrough results in many challenging domains. To continuously improve these results and approach human abilities in a broad variety of domains, the complexity of the network (i.e., its depth) increases. Most of these applications are run on data-center-class servers, for which processing speed and energy consumption are primary concerns. For those 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 CNNs requires flexible ASICs, such as the Google TPU [1], which are, however, less efficient than theory would predict, for example because they support only a few numerical data types.

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

Since network depth and complexity increase, mapping a network on a single FPGA in most of the cases fails to meet performance requirements and would benefit from a multi-FPGA implementation. The problem that we are addressing is as follows. We are given an application modeled as an interconnection of tasks, each with various implementation options with varying performance, memory bandwidth, energy and resource requirements. We would like to statically or dynamically allocate resources to these tasks to optimize various measures of performance, such as throughput, energy per operation, and so on. Platforms like the CPU and the GPU use various kinds of schedulers (Operating System scheduler on the SW side, thread and instruction schedulers on the HW side) for this purpose at compile time or at runtime. The goal of this thesis is to design a compilation-like resource allocator for multi-FPGA acceleration. We devised and implemented

an efficient and accurate optimization framework for the allocation of task-level pipelined applications (like Convolutional Neural Networks and Deep Neural Networks) to multiple FPGAs, with the twofold goal of maximizing the application throughput and minimizing the power consumption, under resource and off-chip memory bandwidth constraints. The target Multi-FPGA platform consists of AWS F1 instances with up to eight Virtex Ultrascale+ FPGAs.

First, we implemented in synthesizable C++ and optimized using HLS directives the computing kernel for each and every layer of large CNNs, such as AlexNet, VGG, YOLO, ResNet, and large DNNs, such as Transformer variants. Then, using SDAccel, we implemented individual kernels in hardware using one Compute Unit (CU) for each layer, and orchestrated their execution on the FPGAs by a host code written in OpenCL and executed by the CPU of the AWS board. This allowed us to profile each kernel and get resource and memory bandwidth usage, working frequency, and execution time, which later become the input data of the optimization problem. We provide a model that covers the whole application execution, and includes: 1) input data transfer time from the host CPU to FPGA DDR memory (dynamic RAM), 2) data transfer time from FPGA DDR memory to the FPGA on-chip memory (static RAM), 3) the actual kernel computation, 4) data transfer time from FPGA on-chip memory to FPGA DDR memory, and 5) data transfer from the FPGA DDR memory to the host CPU. This model can be used to mathematically formulate a complex Mixed-Integer Non-Linear Programming (MINLP) optimization problem, which can be solved using a commercial MINLP solver [2]. However, using a MINLP solver is very slow, since the problem is NP-complete [3]. To accelerate the optimization process, we provide a fast heuristic method using a Geometric Programming (GP) [4] solver and an allocator. Not only it can return the solution in a matter of seconds, instead of running several hours or days when using the MINLP solver, but it also offers better results than those returned by the solver when its run time is limited for practical reasons.

Second, we developed another optimization framework to find the solution with minimum power consumption for a given throughput. This model is aimed at data center applications, where energy and cooling costs are significant. To optimize the power consumption we provide a power model on top of the performance model. This model includes the power consumption in different phases: 1) data transfer between host CPU and FPGA memory, 2) data transfer between FPGA and DDR, 3) FPGA computation. Given a throughput constraint, the model will return the best number of parallel number of powered-on FPGAs and their clock frequency and generates the most power-efficient bitstreams to program the FPGAs. This model can also lead to the formulation of another Mixed-Integer Non-Linear optimization problem, which can also be solved using a MINLP solver. We compared the solution obtained by the solver with one that simply clock gates the fastest implementation and one that uses frequency scaling: our method always uses less power. However, a MINLP solver can be very slow especially for design space explorations which

need to run the solver several times. Therefore, we provide two different heuristic methods. One of them still uses the MINLP solver but in a reduced exploration space; the other one uses a greedy allocation. Both heuristic methods can be a few orders of magnitude faster than the MINLP solver.

Also for power optimization, we use AlexNet, VGG and Transformer networks to verify our model. The experimental results show that our approach can find the best solution compared to both 1) applying frequency scaling to optimize power under a throughput constraint starting from a fast configuration, and 2) replicating a slow configuration on multiple FPGAs.

Acknowledgements

This work would not have been possible without the help and support of many wonderful people.

A few people played a key role in helping me with my research work. Prof. Mario Casu was my supervisor during my Master's thesis. I was really interested in the research work afterward, so I applied Ph.D. in Polytechnic of Turin. Prof. Casu introduced me to prof. Luciano Lavagno, they gave me an opportunity to pursue the work I love. I am so lucky to have them as my supervisors. They taught me how to do research work, told me which topics are more promising, gave the total freedom to choose the topic, and guided me to write articles in every single step so I could present my work in a more clear way. I would like to thank them for their patience and constructive suggestions. Of course, their help was not only limited to my work, their wisdom and enthusiasm affected me in many aspects. I also would like to thank prof. Jordi Cortadella and prof. Mihai Lazarescu for their advice and assistance with my work. They supported me in many ways during the course of my Ph.D.

Many thanks go to the referees of this thesis prof. Dirk Koch and prof. Davide Quaglia who spent their valuable time reading my thesis and giving suggestions to make my thesis more clear and insightful. My thanks also extended to the other committee members prof. William Fornaciari, prof. Roberto Passerone, and prof. Mihai Lazarescu for generously offering their time and support.

I also would like to thank my colleague and friend Osama Bin Tariq whom I collaborate on another project. I still remember we worked until after midnight to meet a deadline. Also how I ruined his trip to London by continuously bugging him to change the paper which later got rejected (but eventually got published). Also, Liang Ma, a dear friend, gave me great help when I joined the group. He always encourages me and teaches me to believe in myself.

I would also like to take this opportunity to acknowledge the support of my group mates (past and present) at the High-level synthesis group. It has been a pleasure working with them. They have been a great source of ideas, research tips. Even doing some other work, such as cooking Italian, Pakistani cuisine, they always managed to keep their wonderful sense of humor.

I am extremely grateful to my parents for their love, prayers, caring, and support throughout the years of my study. Also, I express my thanks to my sister and brother for their support. This accomplishment would not have been possible without them.

I would like to dedicate this thesis to my loving parents

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Multi-kernel Applications

Many data center applications are organized as sequences of sub-tasks, called *kernels* in the following, which are organized roughly as a pipeline. For example, database applications can be organized as a pipeline of classical SQL operators (select, join, . . .) or as map-reduce pipelines, financial algorithms can be modeled as random number generators followed by Monte Carlo simulation steps, and so on. In this thesis we focus on a specific class of such applications that has become very popular in recent years, namely Artificial Intelligence (AI) algorithms. However, the techniques developed in this thesis can be applied much more broadly than the illustrative examples that we use in our work.

We chose AI because it allowed us to focus on the *modeling of the optimization problem*, rather than on the modeling of the application itself, since the synthesizable C++ code for most CNN and DNN applications is freely available, and because it has witnessed monumental growth in recent years. Researchers and enthusiasts alike, work on numerous aspects of the field to make amazing things happen. Among all AI fields, Machine Learning (ML) is one of the most studied and used to solve different classes of problems.

ML comprises a wide variety of algorithms, among them Deep Learning (DL) is the the most promising one. The key concept behind DL is that it uses multiple levels of data features, hence the name deep. Starting from the raw input data, several subsequent levels of features with different abstraction levels are extracted and learned. Abstraction levels grow from low to high towards the outputs of the network.

Artificial Neural Networks (ANNs) are a subset of Machine Learning models. ANNs can be shallow or deep. In the latter case, they are referred to as Deep ANNs (Deep Neural Networks, DNNs). Convolutional Neural Networks (CNNs)

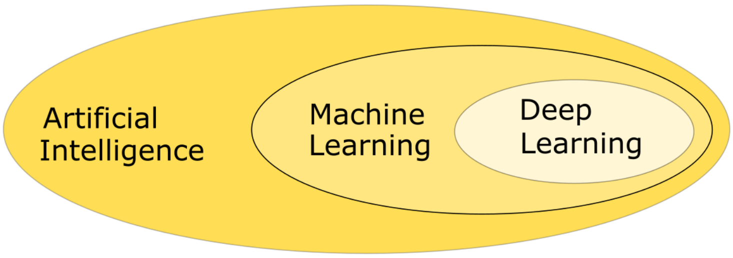


Figure 1.1: Relation between AI, ML and DL.

are a typical class of DNNs. They have been used for both image recognition [5] [6] [7] and natural language processing [8]. They can thus be used to process live data for traffic surveillance cameras, identify people in pictures, transcribe voice and analyze text to perform "sentiment analysis" (for customer support or to improve user experience on social networks). To continuously improve these results and approach human abilities in a broad variety of domains, the DNNs depth increases, thus requiring more resources, and more design effort to optimize performance over a deep task-level pipeline composed of multiple kernels.

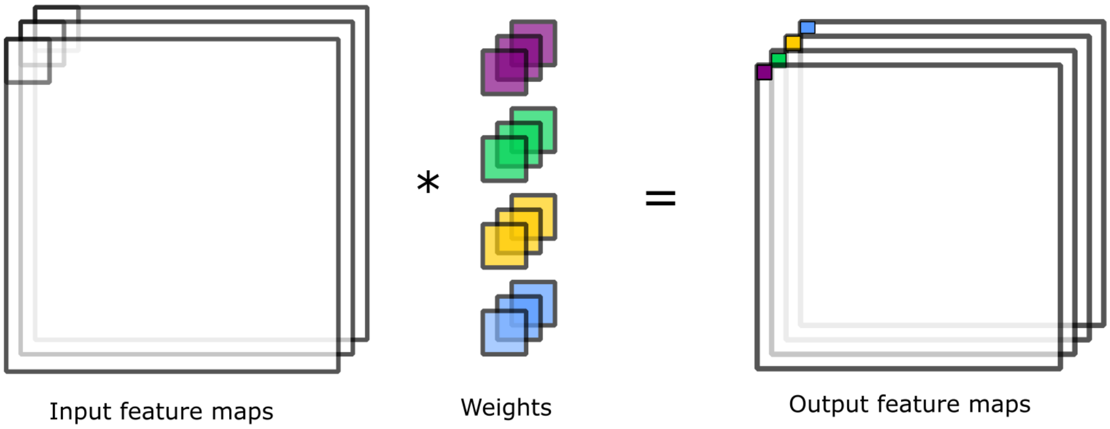


Figure 1.2: Convolution layer.

A typical CNN algorithm is made of several layers. All the layers are connected in a sequential order. The layers are based on a few key components, convolution, rectification, pooling, and fully connected layers. Among all the layers, the most computationally significant layers are convolution and fully connected layers. Figure 1.2 shows a multi-dimension convolution operation. The input feature map has three channels, this is the case for image recognition where the three channels represent R, G, and B colors, respectively. The convolution uses four filters, each of them having three channels to match the input feature maps. Each of the filters and the sliding window on the input feature map are having a point-wise multiplication and addition to produce one pixel of the output feature map. In a typical CNN, following the convolution layer is rectification. It introduces the non-linearity to the network allowing the model to learn faster and perform better. Another typical CNN layer is pooling, which can be used to form a down-sampling. The most used pooling methods are max and average pooling. The last layers in a CNN are typically the fully connected ones, which generate a vector whose size is equal to the number of classes that the network can discriminate. Then the classifier will return the class which has the highest probability. Since we only focus on the inference phase of the network, the training part is omitted here.

CNNs are known to be computation-intensive. For example, AlexNet [5] has 0.7 GFLOPs, VGG [6] has 19.6 GFLOPs. These applications are used for computer vision to perform *inference*. It is clear that such kind of applications have very strict power and performance requirements. In this thesis, we focus on energy efficient CNN accelerations.

1.1.2 Heterogeneous Computing Systems

To achieve high performance with good power and energy consumption, choosing a suitable computing system platform is vital. This section explains the characteristics of different heterogeneous computing systems.

A homogeneous computing system, composed of a group of CPUs has been the preferred solution to build High Performance Computing systems and data centers for a while. However, it is no longer able to achieve the remarkable performance demanded in modern data centers because Single Instruction Single Data CPUs are no longer increasing performance at the same rate as Single Instruction Multiple Data GPUs and Vector Processors, or as reconfigurable platforms such as FPGAs, and they are also much less memory efficient [9]. The solution to this issue is provided by heterogeneous computing systems.

They are called heterogeneous because the co-processors are different from the host device, e.g., they have different instruction sets or architectures, and the programming languages and environments are different. The performance and energy efficiency can be achieved by the co-processors with specialized processing capabilities to handle particular tasks. The co-processors can be the Graphic Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), and any other Application-Specific Integrated Circuits (ASICs). The co-processors communicate with the host processors through the Peripheral Component Interconnect Express (PCIe) bus. Each processor has an independent memory. The co-processors may communicate with each other via the same PCIe bus, or via other dedicated protocols (e.g. Aurora for Xilinx FPGAs).

Graphic Processing Unit

GPUs are specifically designed integrated circuits originally used to process graphical information such as images and videos. Currently, they are widely used as accelerators for parallel computations such as training machine learning algorithms. The architecture of the GPUs contains many computation cores also named Algorithm-Logic Units (ALUs) managed by a single control unit.

The GPU can be programmed in CUDA, a proprietary programming language that provides C/C++ syntax rules-based language and programming environment, or the very similar (but open) Open Computing Language (OpenCL) which is a framework to compile programs for executing on heterogeneous platforms. So the GPU can be easily used by software developers.

However, it has been shown that these platforms are not very efficient with respect to the energy consumption for many kinds of applications including the machine learning algorithms that we considered in this thesis, mostly because (1) they have a datapath that supports only a few fixed data widths (e.g. FP32, FP16 and int8) and (2) their memory subsystem is very flexible and powerr-hungry. This issue has been addressed recently by using reconfigurable hardware platforms as accelerators.

Application-Specific Integrated Circuit

An ASIC is an integrated circuit customized for particular tasks rather than intended for general-purpose use. So the ASIC has limited programmability as an accelerator in a heterogeneous computing system. However, a well designed ASIC chip usually achieves the best performance and energy efficiency for the particular tasks running on it. ASICs can provide the best energy efficiency, but the continuous evolution of DNNs requires flexible ASICs, such as the Tensor Processing Unit (TPU) [1], which are, however, less efficient than theory would predict, mostly because they are also limited to a fixed (typically systolic) computational graph, and to a few (e.g. FP16 and int8) data widths. TPUs have been designed from the bottom up to allow faster execution of applications. However, TPUs are only good at performing dense vector and matrix computations and are specialized in running very fast programs based on Tensorflow. They are very well suited for applications dominated by matrix computations and for applications and models with no custom TensorFlow operations inside the main training loop. That means that they have lower flexibility compared to CPUs and GPUs and they only make sense to use them when it comes to models based on TensorFlow. FPGA on the other hand can be used to perform high-performance matrix computations. Moreover, it is much more flexible compared with TPU. It can be programmed to perform other kinds of computations.

Field-Programmable Gate Array

An FPGA is a programmable integrated circuit that exploits a reconfigurable spatial computing architecture for massive parallelism rather than the Instruction Set Architectures (ISAs). On modern FPGAs, such as the Stratix from Altera and the Ultra-Scale+ and Versal families from Xilinx, there are up to millions of Configurable Logic Blocks (CLBs) and Flip-Flops, megabytes of on-chip the Block RAM (BRAMs), hundreds of multiply and accumulate units (DSPs), and many other dedicated hardware blocks, including ARM Cortex processors [10]. These CLBs can be connected via a hierarchy of reconfigurable interconnects (configurable wires) to perform complex combinational functions and sequential functions. The integration of the DSPs makes the modern FPGAs also eligible for floating-point computing acceleration.

1.1.3 FPGA Design Methodology

FPGAs are a promising option for CNN acceleration in data centers, 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.

The Register Transfer Level (RTL) models are the predominant starting point for standard design flows for FPGAs. These models are written in a Hardware Description Language (HDL), and then are synthesized, placed and routed by Electronic Design Automation (EDA) tools. However, this traditional design flow is losing steam. On one hand, it is very time-consuming, tedious, and error-prone to code complex algorithms since it usually needs thousands of lines. This characteristic limits flexibility. To test the correctness of the model, the user needs to write a complex testbench. On the other hand, the standard software development flow, based on the principle of “write once, run anywhere” is attractive for hardware designers. Both Altera/Intel and Xilinx promise software-like development for applications that are entirely written in a high-level language and are then compiled and synthesized for heterogeneous CPU-FPGA platforms. This software-like design flow is named high-level synthesis (HLS). HLS design flow can dramatically reduce the design and verification costs, essentially eliminating the need to model the design at RTL.

Given an algorithm modeled in a high-level language such as C, C++ or OpenCL, several optimizations can be applied to improve its performance (and resource utilization) on an FPGA. The optimizations can be done by using directives in HLS. The most used directives are:

- *Loop pipelining* starts new iterations of a loop before the completion of the previous ones. It is one of the best options for loop optimization in HLS, since it usually boosts the performance at a very low cost [11] [10].

- *Loop unrolling* creates multiple copies of the loop body to be executed fully in parallel if their is no dependency among the iterations. In some cases it can achieve even more performance than by means of pipelining, but typically at a huge resource (i.e., area) cost [11].
- *Compute unit* is another mechanism to increase parallelism that is similar to loop unrolling, but at a higher level.
- *DATAFLOW*. Computational processes in dataflow micro-architectures are controlled by the availability of the input data rather than a centralized finite state machine (FSM).

1.1.4 Multi-FPGA Platform

As network depth and complexity increase, single-FPGA designs cannot always meet performance requirements. Multi-FPGA can be a promising option for accelerating high computation-intensive data-center applications to achieve high performance. For these reasons, cloud providers like Amazon (AWS) offer Elastic Compute Cloud (EC2) F1 instance which includes Virtual Machines coupled with multi-FPGA platforms to accelerate data-center applications with GPU-like performance but consuming less energy. Amazon EC2 F1 instance is also the platform we use for our experiments in this work.

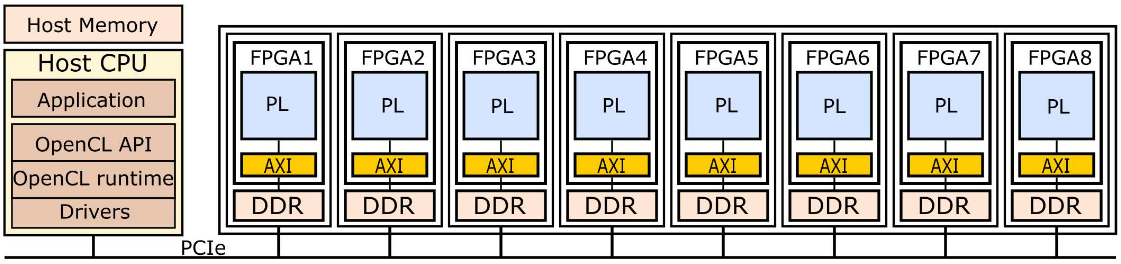


Figure 1.3: Architecture of the Amazon EC2 F1 instance.

Figure 1.3 shows the architecture of Amazon EC2 F1 instance. It has eight Xilinx UltraScale+ FPGAs, each equipped with local DDR DRAM and connected via the PCI express (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. Table 1.1 shows the specifications of the UltraScale+ device **xcvu9p** adopted by the Amazon EC2 F1 instance.

The host CPU has been used to program the FPGAs once the bitstreams are ready, it also works as a control unit to handle the data transfer from the host CPU to the FPGA, as well as the FPGA execution. The kernels on the FPGAs can be

Device	BRAM	URAM	DSP	LUT	LUTMem	REG
xcvu9p	2088	960	6837	1110146	575398	2264435

Table 1.1: Xilinx UltraScale+ xcvu9p device specifications.

synchronized by the host CPU and all the kernels can be executed concurrently. All these requirements can be passed to FPGA compilation environments like Xilinx SDAccel [12] in the case of the Amazon EC2 F1 instance.

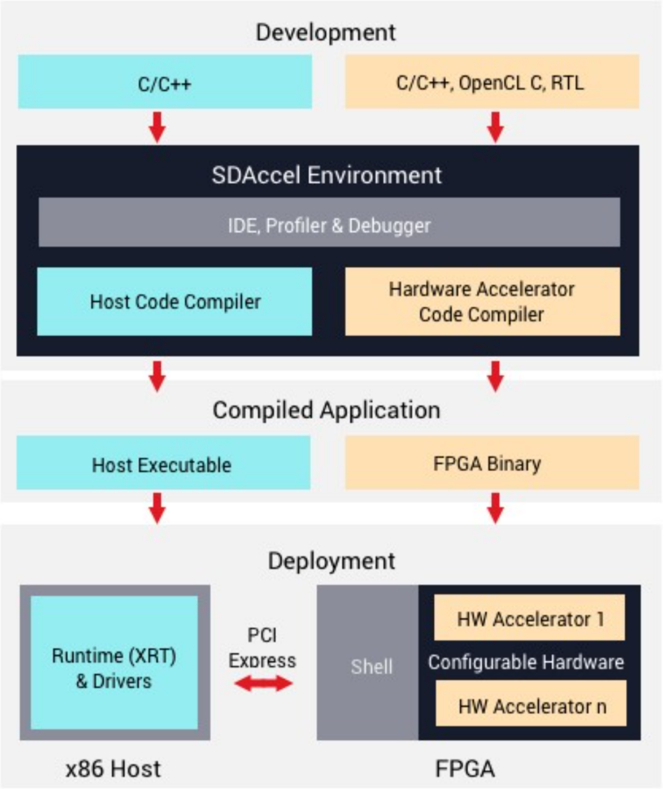


Figure 1.4: Flow of using SDAccel[12].

The SDAccel environment is an integrated development environment for applications targeting AWS F1 instances and other FPGA-as-a-Service offerings. Figure 1.4 shows the flow of using SDAccel for your design.

It provides a familiar software development flow with:

- An Integrated Development Environment (IDE)
- A profiler to guide application optimization
- Compilers for host & FPGA-accelerated code

- Emulation flows for rapid development and debug
- Automatic communication between software and hardware

The host application is developed in C/C++ and uses standard OpenCL API calls to interact with the FPGA-accelerated functions which can be modeled in either RTL, C/C++, or OpenCL. This provides familiar entry points for hardware designers and software engineers alike.

The SDAccel IDE provides all the features of a standard software development environment: optimized compiler for host applications, cross compiler for the adaptable hardware, a robust debugging environment, and profilers to identify performance bottlenecks and optimize the application.

The Xilinx runtime (XRT) and board-specific shells automatically manage communication between the FPGA accelerators and the host application. The software developer does not need to implement any of these connection details.

1.1.5 Goal of the Thesis

Thanks to the Xilinx SDAccel design environment, users can easily program the FPGAs starting from models written in C, C++ and OpenCL. However, the Vivado HLS tool that is used by SDAccel only deals with micro-level resource allocation. The global-level resource allocation is totally left to the programmer, who is in charge of defining the number of Compute Units that are instantiated for each kernel, in order to match the throughput of each pipeline stage while satisfying resource and memory bandwidth constraints.

Figure 1.5 shows a task level pipeline model that has three kernels executing one after another. If we pipeline it at a global level, all the kernels can work concurrently, the throughput will be highly increased. The initiation interval is determined by the slowest kernel. The throughput can reach its maximum if we balance the resource allocation in a way that all the kernels have the same execution time shown in Figure 1.6. To do so, SDAccel allows users to instantiate multiple copies of each kernel, called Compute Units (CUs). The workloads can be equally distributed on the multiple CUs to reduce the execution time.

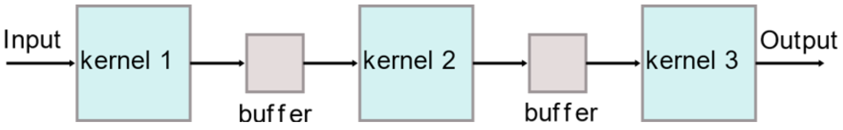


Figure 1.5: Task-level pipeline model.

However, there are no available tools that can be used to accelerate applications on multi-FPGA platforms to maximize throughput at the global level. Our goal in this work is to develop an optimization model that can find the optimal level

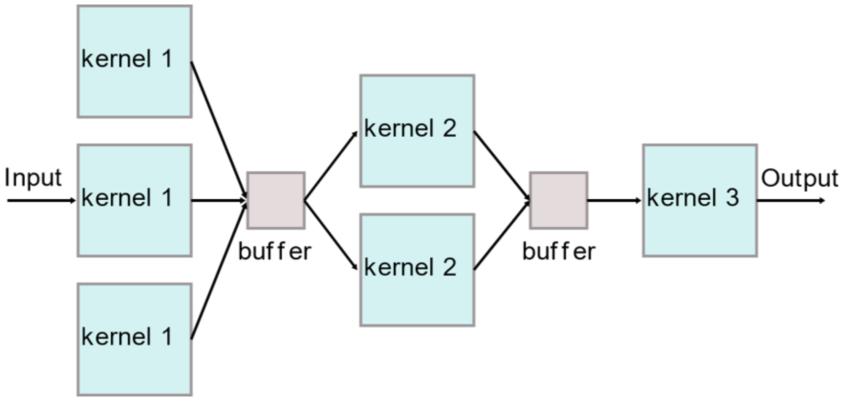


Figure 1.6: Resource balanced task-level pipeline model.

of parallelization as well as the allocation of these multi-kernel applications on a Multi-FPGA system. By giving the number of FPGAs, the available resources, and a set of constraints, the model will return the solution with the maximum throughput, indicate the value of the throughput, on which FPGA the kernels should be allocated after proper parallelization obtained using multiple Compute Units (CUs) for each kernel, and the number of resources distributed to each kernel.

Similarly, we also develop another optimization framework to find the solution with minimum power consumption for a given throughput. It should be able to determine the number of FPGAs in-use, the number of CUs for each kernel, and their allocation on different FPGAs in a multi-FPGA platform.

1.2 Overview of the Contributions

The contributions of this thesis are included in two areas: the throughput optimization model for CNNs on multi-FPGA platform and the power optimization model. Regarding the throughput optimization model, we consider that the layers of CNNs work concurrently in an efficient way by balancing the resource usage for each layer; the resource organization that obtains the maximum throughput can be achieved using a state-of-the-art solver or a heuristic method. Similarly, the latter contribution concerns a model to optimize the total power consumption given a minimum throughput as a constraint. It includes both the power consumed during the data transfer as well as the one spent in computation on the FPGA. The solution of an optimization problem will determine the number of FPGA in use and their working frequency, other than the resource allocation.

More precisely, the contributions of this thesis are the following:

- **Performance optimization model.** 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 (CUs) 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.

  - **MINLP solver.** The optimization problem has been solved initially using a Mixed-Integer Non-Linear Programming (MINLP) solver [2].
  - **Heuristic Method.** 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 Geometric Programming [4] 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 the problem size is small, and it returns much better results for larger problem sizes.

- **Power optimization model.** We have proposed a power-performance optimization method to optimally configure a multi-FPGA platform running multi-kernel pipelined workloads. Given an Initiation Interval (II) target, the solution provides an optimal allocation of the best number of CUs for each kernel so as to minimize the overall power consumption. Compared to applying frequency scaling to reduce both II and power starting from a fast configuration, or to replicating a slow configuration on multiple FPGAs, our solution provides a much more effective way of saving power.

  - **MINLP solver.** The optimization problem has been initially solved using a MINLP solver. However, it takes too long to find an optimal or near optimal solution.
  - **Heuristic methods.** We then proposed two different kind of heuristic methods to increase the efficiency. The first heuristic method still uses a MINLP solver but with a reduced exploration space, thus increasing the speed. Similar to the first one, the second heuristic method also uses the same method to reduce the exploration space, but instead of using the MINLP solver, it uses a greedy allocation.

  Experimental results shows that the first heuristics constrains the exploration space to significantly reduce the run-time, while achieving the same or even better results than the exact algorithm. Moreover, the second heuristics is thousands of times faster than the exact algorithm.

1.3 Organization of the Thesis

This thesis presents a collection of the work done in the field of electronic design automation (EDA) for multi-FPGA design of multi-kernel applications. The work is composed of seven chapters and their organization is as follows:

- Chapter 2 covers the state-of-the-art.
- Chapter 3 provides a simplified performance optimization model that only considers the execution time of the kernel.
- Chapter 4 proposes a more detailed method to map multi-kernel applications on multi-FPGA platforms to maximize application throughput. We try to use Mixed-Integer Non-Linear Programming (MINLP) solver to find the optimal solution. 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.
- Chapter 5 discusses the way to minimize the power consumption for data center applications on Amazon EC2 F1 instance. we propose to upload at run-time the best power-optimized CNN implementation for a given throughput constraint. The off-line optimization model can be solved using a Mixed-Integer Non-Linear Programming (MINLP) solver, it gives the best number of parallel instances of each kernel, their allocation to the FPGAs, the number of powered-on FPGAs and their clock frequency.
- In chapter 6, we provide two heuristic optimization methods that improve the quality of results of the power optimization model discussed in chapter 5. We use several very large designs to demonstrate that both heuristics obtain comparable results to a MINLP solver when this can find the best solution, and they obtain much better results than the MINLP solver when this cannot find the optimum within a bounded amount of time.
- Chapter 7 concludes the work and discusses the possible future work.

Chapter 2

Related Work

Resource allocation is a well-studied problem for high-performance data centers with heterogeneous hardware (CPUs with Graphical Processing Unit (GPU) or FPGA accelerators). Yet, the context of multi-FPGA platforms still requires investigation, as the review of the literature that follows shows. Here we contrast the previous work in this field with ours, and highlight the most important differences between what has been proposed by other researchers and our own work. We divide the literature analysis in two parts, which correspond to the two main contributions of this thesis.

2.1 Performance Acceleration

The community interested in compilers for parallel architectures faced a similar problem when mapping streaming applications to multiprocessor systems and accelerators. Indeed, [13] defines three levels of parallelism (task, data and pipeline) that are also exploited in our underlying execution model (tasks are called "kernels", data parallelism is exploited both at the CU level and the loop unrolling level within a CU, and innermost loops are pipelined). Their compiler, based on the StreamIt language, is aimed at processors (the RAW machine) rather than FPGAs. Moreover, it makes only heuristic choices for allocation. Similarly, [14] uses multiple process instances, but focuses only on process replication and FIFO allocation, while we include resources as a primary aspect of our cost function and consider array-based communication, rather than FIFO-based. Array-based is a more natural programming model, because it is supported by languages like C, C++ and OpenCL, and it requires fewer changes to legacy code, without complex logic for forking and joining data to and from data parallel CUs. More recently, [15] includes, like in our case, an explicit memory model, but solves the problem heuristically with a clustering algorithm (using ILP only as a reference), while we start from a GP relaxation for our heuristic.

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 [16, 17, 18]. 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. [19] schedules a task-parallel Static Dataflow Graph with multiple CU instances, leading to a very efficient scheduling formulation as a Set of Difference Constraints. However, it is also limited to FIFO-based communication and it does not consider multi-FPGA allocation and the resulting trade-offs.

In the multi-FPGA side, the authors [20] 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, [20] also adopts an on-board data reuse scheme to minimize the external memory access time.

The Brainwave project [21, 22] developed by Microsoft is designed for real-time AI, which means the system can ingest a request as soon as it is received over the network at high throughput and at ultra-low latency without batching. They did a very good job to allow users without hardware expertise to automatically deploy and accelerate state-of-the-art DNN models in real-time and at low cost. However, their main focus is the recurrent neural networks for text-driven applications at Microsoft. This type of application is bandwidth-intensive and more difficult to accelerate than CNNs. To solve the challenges, they exploit model parallelism and store all the parameters on-chip. When an FPGA's on-chip memory is exhausted, the system will use more FPGAs to allow all the weights to be stored in on-chip memory. They also provide a quantization mechanism to trim the bitwidth of the weights. Different from the Brainwave project, we are more interested in throughput, which is a more significant figure of merit for CNNs or DNNs used for image classification in a non-real-time context. Our work is focusing on accelerating throughput-demand multi-kernel applications. All the kernels in a multi-kernel application are working concurrently.

In [23], 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 initiation interval (II) of the pipeline in [23] 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 [23] the consecutive kernels are forced to be allocated on the same FPGA, while our model does not force that. Finally, [23] does not consider the frequency reduction due to routing congestion when the resource utilization increases, while we consider it.

Similar to our work, in [24] 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 [23], namely that we do not restrict the distribution of CUs to FPGAs to be grouped by stages.

Also in [25], a preliminary characterization of kernels, termed as components, is done before a design-space exploration of a system made of several components is performed. In that work, an application is modeled as a Timed Marked Graph and Petri net theory is used to find the best overall throughput, then imposing a throughput constraint on every process and trying to satisfy it via High-Level Synthesis. However, there is no guarantee that the requested throughput is feasible, hence iterating is needed to explore the entire Pareto-optimal design space. Moreover, the paper does not discuss memory bandwidth nor allocation to FPGAs.

In [26], 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, [26] provides an efficient BLAST algorithm using Dynamic Programming (DP), while we use a Geometric Programming solver and a heuristic allocator to improve the efficiency.

[27, 28] 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.

Finally, Maxeler Technology offers to its users dataflow HPC solutions. The workstations are hybrid computing platforms that are using both CPUs and FPGAs [29]. In order to use the Maxeler system, three basic parts need to be provided by the user: 1) The CPU interface code to handle the data flow, 2) kernels which need to be implemented on FPGA, 3) the manager which handles the internal functionality and the on-board and off-board data movement. Maxeler FPGA [30] dataflow engines run at a few hundred megahertz, and can already beat the performance of conventional CPUs. In addition, dataflow engines are easily able to exploit increasing silicon capacity since performance comes directly from parallelism and can scale linearly with silicon area, without depending on clock frequency increases. In order of using Maxeler DFE, applications should meet the following four criteria:

- BigData. The first advantage of using Maxeler DFEs is that they are able to accelerate the movement of data. So the application has to be a real BigData application.
- Extensive data reuse. The application should use most data more than once.
- Loop structure. The application should have loops that consume most of the processing time. The loops are the portions of the application that are move to the DFE.
- Initial latency. The algorithm must tolerate initial latency.

From [30], we know that the Maxeler FPGAs are the dataflow engines. The application is running on the FPGA with a dataflow mechanism. Similar to their work, we also use FPGA to design accelerators in order to improve the application throughput. However, we are using dataflow at a higher level. Instead of using dataflow inside each kernel, we use dataflow at the kernel level. To use our method, the application should have multiple kernels in order to "see" the advantage. In addition, inside each kernel, the loops are also unrolled and pipelined. This gives another level of parallelization. Moreover, each FPGA in the AWS platform has three Super Logic Regions (SLR). If kernel instances use more than the resources of an SLR, then there will be Super Long Lines (SLL) involved, reducing the working frequency. In our work, each kernel instance is using a small portion of the FPGA, each FPGA can have several kernels allocated on it, and all the kernels are totally independent and they are working concurrently. This will result in less SLR crossing, thus higher working frequency.

2.2 Scheduling and Resource Allocation

For the resource allocation on multi-core systems, a large number of past works are using strip packing or bin packing. In our work, we provide an allocator to assign the kernels to different FPGAs. There are some similarities to the Strip-Packing, Bin-Packing problems. Strip packing [31] problems involve packing items into a single bin of fixed width and infinite height, with the objective of minimizing the total height of the packing within the strip. [32] proposed online scheduling for multi-core shared reconfigurable fabric. They modeled the task queue as a 2D rectangular Strip-Packing problem (2D-SPP) with the additional processor and deadline constraints in order to find the optimal schedule. Unfortunately, 2D-SSP is an NP-hard problem, and finding the optimal solution with a large number of rectangular is not feasible. However, it can provide a good target for the online scheduler.

Bin-Packing problems [33], on the other hand, involves packing items into multiple bins of fixed width and height, so as to minimize the number of bins utilized. [34] and [35] are using Bin-Packing to schedule tasks on multi-core systems. [34] provides an optimization method for the task allocation step for multi-core processors. To complete their model, all the created tasks are mapped to the available processing cores by using Bin-packing heuristics. [35] studies the problem of how to schedule real-time tasks on multi-core platforms to maximize energy efficiency under other constraints, like temperature.

Our allocation problem is similar to the bin packing problem. Each kernel is using a certain amount of resources, hence it is like a rectangle in the bin packing problem. Instead of minimizing the number of bins, we try to fit all the rectangles in a limited number of bins, in addition to satisfying other constraints (e.g. memory bandwidth).

2.3 Multi-FPGA Platform for Emulations

Researchers also use multi-FPGA as emulation platforms [36, 37, 38, 39, 40, 41]. [36] is using multi-FPGA for ASIC prototyping. [37] and [40] are using multi-FPGA for multi-core processor simulations. [38, 39] and [41] are using it as logic emulation platforms for Networks-on-chip.

[36] summarizes a number of issues that should be considered by designers, including partitioning the netlist on multiple FPGAs. The authors suggested that each FPGA should be treated as a block within an overall top-down ASIC design flow, so it can help to localize the effect of design changes, reducing iteration time.

[37] described a multi-FPGA platform to accelerate logic verifications of the Bluegene compute node ASIC, a multi-processor SOC implemented in IBM's CMOS technology. It discussed the challenges including the design partitioning. The authors developed their own partitioning tool to map a DUT onto FPGAs. The tool takes three inputs: 1) the netlist of the DUT, 2) the netlist of the physical hierarchy of the FPGA systems, and 3) a mapping file. The tool analyzes the netlist and writes a complete set of VHDL files contain the appropriate instances from the logical hierarchy. In addition, the tool also automatically generates timing constraints. [40] proposed a method to emulate a 48-core multiprocessor on multi-FPGA. The DUT is mapped onto one or several FPGAs by commercial ASIC/FPGA RTL synthesis tools.

Hung *et al.*in [38] pointed out various challenging problems for logic emulation using FPGA. The DUT will be partitioned into hundreds of pieces, and each of them is allocated an FPGA without exceeding the resource utilization. The objective of partitioning is to minimize the cut sizes.

Abdellah-Medjadji *et al.*in [39] provided an accurate multi-FPGA emulation platform. They try to partition the NoC into K subsets and each one assigned to a reconfigurable device N ($K \leq N$). This algorithm is trying to minimize the total intra-cluster links under the constraint that a cluster must fit on one FPGA chip. Different from [39], in our work, each kernel is a separated subset, and the number of subsets can be higher than the number of FPGAs. More than one cluster can be allocated on the same FPGA. Similarly, Karypis *et al.*in [39] proposed a partitioning algorithm for applications in the VLSI domain. The presented multilevel hypergraph-partitioning algorithm directly operates on the hypergraphs. They developed new multiphase refinement schemes based on the multilevel paradigm. These schemes take an initial partition as input and try to improve them using the multilevel scheme. These multiphase schemes further reduce the run times, as well as improve the solution quality. The coarsening phase is able to generate a sequence of hypergraphs that are good approximations of the original hypergraph. The initial partitioning algorithm is then able to find a good partitioning by essentially exploiting global information of the original hypergraph. Finally, the iterative refinement at each uncoarsening level is able to significantly improve the partitioning quality because it moves successively smaller subsets of vertices between the two partitions.

2.4 Power Efficient Resource Allocation

Power-optimal FPGA design is a broadly investigated field, but mostly focused on single-FPGA designs.

The traditional way of reducing power consumption consists in using frequency scaling and dynamic voltage scaling methods. [42] presents a universal offline self-calibration scheme, which automatically finds the FPGA frequency and core voltage operating limit at different self-imposed temperatures by monitoring design-specific critical paths. [43] investigates the energy reductions possible in commercially available FPGAs configured to support voltage, frequency and logic scalability combined with power gating. [44] presents a method of dynamic voltage and frequency scaling that uses online slack measurement to determine timing headroom in a circuit while it is operating and scale the voltage and/or frequency in response. All these methods focus on low-level aspects, whereas we propose a complementary high-level approach, which can minimize the power consumption by allocating the kernels in a certain way and also determine the working frequency of each FPGA in a multi-FPGA platform.

Tesfatsion *et al.* [45] provide a resource management framework with a hardware scheduler and an optimizer for FPGA-accelerated clouds. Similar to our work, they split workloads into "chunks" run by Virtual Machines on CPUs and sharing FPGA accelerators. But they do not pipeline chunk execution and consider only the FPGA static power.

Zhang *et al.* [46] map pipelined CNN layers to a multi-FPGA platform exploring the design space for optimal performance and energy with dynamic programming. However, they assume constant FPGA power consumption, thus reducing the problem of energy minimization to execution time minimization. Also, they use First In First Out queues (FIFOs) for inter-layer communications, which require in-order production and consumption of activation values. This may be difficult to achieve, and is not supported by current multi-FPGA cloud platforms like Amazon AWS F1 (FPGA-to-FPGA transfers must be mediated by the CPU). On the other hand, we model inter-kernel communication using memory arrays, which is arguably a more general and natural programming model, supported by C, C++, and OpenCL.

The execution model in [13] exploits, like our work, application parallelism at task, data, and pipeline level, but the authors target processors instead of FPGAs. Furthermore, a compiler decides the allocation through heuristic moves, while we solve an optimization problem. A task-parallel static dataflow graph execution model with multiple CU instances is proposed in [19] for FPGA targets, with efficient scheduling formulated as a set of difference constraints. But it does not consider multi-FPGA platforms and optimizes only performance, not power.

For multi-FPGA targets, [47, 48] propose to improve performance by using direct network communication between FPGAs. However, they do not optimize the power of the FPGA clusters, and again this communication model is not offered by current PCIe-based multi-FPGA cloud platforms.

Li *et al.* [49] use a similar greedy resource allocation to the most critical kernel, balancing resource usage until exhaustion, but without minimizing power consumption or considering multi-FPGA allocation. Our model satisfies performance constraints while minimizing the overall, multi-FPGA power consumption. Cong *et al.* [19] proposed a task-parallel static dataflow graph execution model with multiple compute unit (CU) instances, with efficient scheduling modeled as a set of difference constraints, but for single-FPGA targets and optimizing only for performance, not power.

Chapter 3

Simplified Performance Optimization Model for Multi-kernel Applications on Multi-FPGA Platform

In this chapter, we optimize the mapping of high-performance multi-kernel applications, like Convolutional Neural Networks, to multi-FPGA platforms. First, we formulate the system level optimization problem, choosing within a huge design space the parallelism and number of compute units for each kernel in the pipeline. Then we solve it using a combination of Geometric Programming, producing the optimum performance solution given resource and DRAM bandwidth constraints, and a heuristic allocator of the compute units on the FPGA cluster.

The performance model used for the optimization is “simplified” as it does not consider the data transfer time between the FPGAs and the host of the Multi-FPGA platform. This is a reasonable assumption when the amount of data to transfer outside each FPGA is relatively small (e.g., when communication happens mostly locally within a single FPGA) or when the amount of data is independent of the allocation.

This work was previously published in [50].

3.1 Introduction

In this chapter we exploit an OpenCL-like execution model. In this model, an application is typically a linear task-level pipeline of kernels, each kernel being composed of independent Compute Units (CU). Each CU in turn contains loops which can be unrolled and pipelined to offer further parallelization. Kernels communicate among each other and with the CPU-bound "host code" via large buffers allocated in external DRAM. The designer must ensure that CUs do not interfere with each other when writing into these buffer, i.e. CU-level parallelism can be arbitrarily increased via replication. This computational model can also be supported by C++-based synthesis tools (in fact, we model our applications in C++ in order to have better control over loop handling during HLS), and fits very well many datacenter applications, like CNNs or other Neural Networks and Machine Learning algorithms.

However, globally optimizing the throughput of a task-level pipeline of kernels over multiple FPGAs is far from trivial. One must take into account simultaneously:

1. *throughput* matching among multiple kernels, which can be increased or decreased by changing either the number of CUs or the parallelism of each CU (e.g., via unrolling);

2. the amount of *resources and external DRAM bandwidth* used on each FPGA, which increases as more CUs are allocated to them.

The number of choices to evaluate, and hence the designer expertise and effort needed, quickly grows out of control. Note that while this problem superficially resembles the classical pipeline scheduling problem in HLS, the actual model is much more complex, because CUs that implement kernels:

1. have many more implementation choices (e.g., via unrolling or other HLS transformations [25]) than typical Functional Units.

2. have a multi-dimensional cost function including performance, memory bandwidth, and FPGA resources (DSPs, LUTs, FFs, and BRAMs).

In this work, we propose a new optimization method for the implementation of task-level pipelined applications on multiple FPGAs. We assume that all communication is performed via off-chip DRAM, which is essentially the above-mentioned OpenCL inter-kernel communication model. In this scenario, our method can be used to choose how many CUs should be allocated for each kernel. This is a simple option that can be passed to FPGA compilation environments like Xilinx SDAccel, Intel SDK for OpenCL, and so on. While a mix of on-chip and off-chip communication resources would allow the exploration of an even larger design space, they are not yet supported by any of these design environments. Hence their analysis is left to future work.

Our work is fully general, and could be applied (1) to other task-level pipelined applications beyond *CNNs*, (2) to other cloud-based or super-computing FPGA platforms beyond *Amazon Web Services (AWS) F1 instances*, and (3) to other design environments beyond *SDAccel*. However, we use this generally available and well-known trio to demonstrate and quantitatively evaluate our results.

Not all applications can be optimized using the proposed method. This method can only be used for applications where the workload is arbitrarily parallelizable and can be modeled as a pipeline, such as DNNs and some data-center applications, like Financial simulation algorithms, some Database algorithms, video encoding, and decoding. We acknowledge that the method is not efficient for other application types, e.g. finite element methods.

In this work we use two Convolutional neural networks, AlexNet [5] and VGG16 [6]. Note that *our algorithms do not depend at all on the considered networks*, and these two examples are used only for the sake of illustration. Each CNN is composed of several convolutional, pooling, normalization and fully connected layers, and each convolutional layer is mapped to a kernel. As discussed in [51], we use loop tiling to reuse both the input feature maps and the weights. Memory access is optimized by reshaping the input and output feature map arrays and the weight array, to allow burst mode data transfers.

In these applications, throughput (i.e. processed images per second) is the main measure of performance, while *overall latency* (i.e. total pipeline depth) is much less important. Hence we focus on *minimizing the maximum latency among all kernels*, because it determines the Initiation Interval (II) of the pipeline, and therefore its throughput. Note also that memory bandwidth of external DRAM can be a major factor limiting the performance of memory-intensive applications like CNNs. Hence our cost and performance model takes this aspect explicitly into account.

Our flow starts from CNN models which have already been partitioned into kernels and individually optimized for FPGA implementation. Then we collect cost, memory bandwidth, and performance (throughput and latency) data from each kernel, by running several versions of its CUs, with varying degrees of parallelism, on an AWS F1. We then use these values to formulate an optimization problem that is discussed in Section 3.2.1 and models the multi-kernel multi-FPGA resource- and bandwidth-constrained allocation problem. This problem can then be solved:

1. either directly by a Mixed-Integer Non-Linear Programming (MINLP) solver, to provide an exact solution in a potentially very long execution time.

2. or indirectly by combining the power of a Geometric Programming (GP) solver, which is followed by an efficient integer relaxation of the problem variables, with a novel allocation algorithm that:

    - discretizes the result of the GP solver, and

- tries to cluster CUs for a kernel on the same FPGA, to simplify the communication coordinated by the host code.

The second method achieves essentially the same level of optimality as the MINLP solver (whenever the latter is able to complete), in a fraction of the time.

We designed our GP model and allocator to optimize the assignment of Compute Units on multiple FPGAs while keeping into account the limitations of modern FPGAs (e.g. the maximum DRAM bandwidth), so that it can handle the large size of typical state-of-the-art CNN applications. Our contributions are:

1. The definition of the multi-FPGA CU allocation problem for linear kernel pipelines and its constraints.

2. The definition of a Non-Linear Programming model for that problem, and its solution both (1) by an exact (very expensive) MINLP solver and (2) by a GP solver, finding an optimal non-integer solution, followed by an allocator aimed at minimizing the spreading of CUs of one kernel to multiple FPGAs.

3. The analysis of their result quality for two large CNN applications, implemented on large multi-FPGA AWS F1 instances.

As mentioned, we are leaving the generalization to (less common) non-linear pipelines and to (not yet available from industrial design environments) on-chip and off-chip communication mechanisms to future work.

This is the organization of the chapter. We define the optimization problem and our heuristic in Sec. 3.2. Experimental results are reported in Sec. 3.3 and conclusions in Sec. 3.4.

3.2 Multi-FPGA Optimization

We consider an application as a set K of kernels organized in a linear pipeline. As mentioned above, CNNs represent a relevant example, in which the kernels are the convolutional, pooling and normalization layers[1]. Each kernel workload is assigned to one or more *compute units* (CUs) that operate concurrently. The kernels communicate through the host CPU. Since the control unit on the CPU side is quite efficient, we do not consider the CPU time in our model. Application throughput is the inverse of the pipeline initiation interval (II), which depends on the execution time of the slowest pipeline stage.

[1] Some max-pooling layers are merged 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.

Let us define WCET_k the worst case execution time of kernel k obtained with only one CU. We consider kernels that are inherently parallel and for which the execution time ET_k scales proportionally to the number N_k of CUs for that kernel:

$$\mathrm{ET}_k = \frac{\mathrm{WCET}_k}{N_k}, \ \forall k \in K \tag{3.1}$$

$$\mathrm{II} = \max_{k \in K} \mathrm{ET}_k. \tag{3.2}$$

To minimize II it is necessary to find the optimal value of N_k under specific constraints. We consider FPGA resource and memory bandwidth constraints, but we do not consider (yet) power constraints.

Table 3.1: Notations used in the model

Notation	**Description**
K	set of kernels
k	index of kernels, 1,2, ..., $\lvert K\rvert$
f	index of FPGAs, 1,2, ..., F
$\mathrm{WCET_k}$	constant; latency of kernel k with one CU
$\mathrm{ET_k}$	variable; latency of kernel k with N_k CUs
R_k	constant; FPGA resources used by one k's CU
B_k	constant; FPGA bandwidth used by one ks CU
R	constant; resource limitation in one FPGA
B	constant; bandwidth limitation in one FPGA
$n_{k,f}$	variable; CUs of kernel k allocated to FPGA f
N_k	variable; sum of $n_{k,f}$ over all the FPGAs
ϕ_k	variable; spreading function of kernel k
ϕ	variable; global spreading function
II	variable; initiation interval

As an additional design exploration knob, we can deploy an application onto one or more FPGAs of a multi-FPGA board like the AWS F1 instance, which includes eight Xilinx UltraScale Plus FPGAs. This is also the FPGA platform where we run our experiments. In this platform, a host CPU orchestrates the execution of the kernels. Figure 3.1 shows the architecture of the F1 instance. Table 3.1 summarizes variables and constants used in the problem.

The design goal is therefore not just determining the optimal N_k, but also how these CUs are allocated on F FPGAs. If we define $n_{k,f}$ as the CUs of kernel k on FPGA f, we have

$$N_k = \textstyle\sum_{f=1}^{F} n_{k,f}, \forall k \in K. \tag{3.3}$$

Since we assume a uniformly accessed global memory, in our model a kernel execution time depends on the number of CUs but not on where they are allocated.

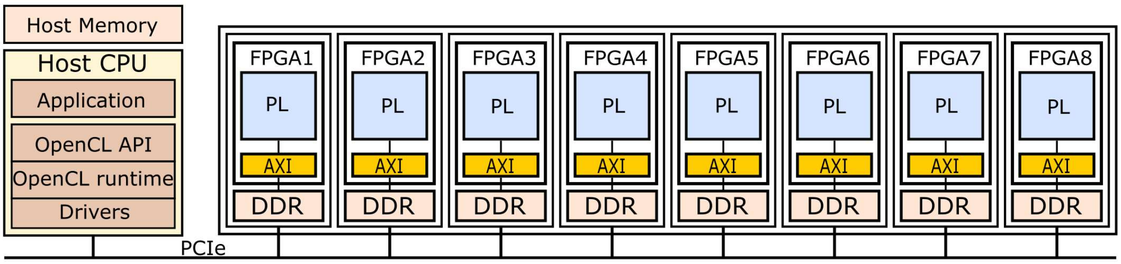


Figure 3.1: Architecture of the Amazon Web Service (AWS) F1 instance.

However, keeping the CUs of a kernel in the same FPGA simplifies the host code (each pair of kernels needs only one buffer to communicate). To account for this, we introduce a *spreading* function that is minimal when all CUs of a kernel are allocated on one FPGA:

$$\phi_k = \textstyle\sum_{f=1}^{F} \frac{n_{k,f}}{1+n_{k,f}} \quad \forall k \in K. \tag{3.4}$$

To minimize the global II and the spreading of the CUs we formulate the optimization problem shown in the following.

3.2.1 Problem Formulation

We can combine II and spreading objectives linearly with two weights α and β into a single goal function g to minimize. The problem is then formulated as a non-linear problem with both integer and real variables:

$$\begin{aligned}
\text{minimize } g &= \alpha \cdot \text{II} + \beta \cdot \phi && (3.5)\\
\text{subject to} & && \\
\text{II} &\geq \text{ET}_k, \quad \forall k \in K && (3.6)\\
\phi &\geq \phi_k, \quad \forall k \in K && (3.7)\\
N_k &\geq 1, \quad \forall k \in K && (3.8)\\
\textstyle\sum_{k=1}^{|K|} n_{k,f} R_k &\leq R, \quad f = 1{,}2, \ldots, F && (3.9)\\
\textstyle\sum_{k=1}^{|K|} n_{k,f} B_k &\leq B, \quad f = 1{,}2, \ldots, F && (3.10)
\end{aligned}$$

The constraint (3.8) guarantees at least one CU per kernel. In (3.9) and (3.10), R_k and B_k are resource and memory bandwidth utilization, respectively, of each CU of kernel k in each FPGA, their sum over all kernels should not exceed R and B, the total resources and bandwidth of a single FPGA.

3.2.2 MINLP solver and GP solver

The optimization model formulated in (3.5)-(3.10) is a typical MINLP model. It involves integer and non-integer variables. Some of the constraints are not linear.

Mixed-integer Non-linear Programming

Mixed-integer nonlinear programming (MINLP) combines the modeling capabilities of mixed-integer linear programming (MILP) and nonlinear programming (NLP) into a versatile modeling framework [52]. Furthermore, by using both linear and non-linear functions it it possible to accurately model a variety of different phenomena. However, MINLP is often considered as a "difficult" class of optimization problems. There are few methods can be used to solve MINLP problems. Branch and bound is proved to be an approach to solve MINLP problems by Gupta and Ravindran [53]. It solves the MINLP problems by relaxing the integer restrictions of the original problem and solving continuous NLP relaxations. Obtaining a tight continuous relaxation is of great importance with branch and bound to avoid large search trees. This is also the reason why, when the problem size increase, the optimization time increase exponentially.

Geometric Programming

A geometric program (GP) is a type of mathematical optimization problem characterized by objective and constraint functions that have a special form. A geometric program is as follows:

$$\textit{minimize} \quad f_0(x) \tag{3.11}$$

$$\textit{subject to} \quad f_i(x) <= 1, i = 1, ..., m, \tag{3.12}$$

$$g_i(x) = 1, i = 1, ..., p. \tag{3.13}$$

where f_i are posynomial functions, g_i are monomials, and x_i are the optimization variables. Moreover, all the variables are positive, i.e., $x_i > 0$. Since it relies on convex optimization, this method is very fast [54]. By relaxing the integer variables in (3.5)-(3.10), it also can be solved using GP solver as a preliminary step to speed the optimization time, then discretize the variables to integer.

3.2.3 Heuristic Solution

As we mentioned in Sec. 3.2.2, the optimization problem formulated in (3.5)-(3.10) can be solved by a Mixed-Integer Non-Linear Programming (MINLP) solver. This can lead, however, to a very long optimization time for designs with many kernels and FPGAs. Consider, for instance, that the VGG-net convolutional neural network with 20 layers spread on 8 FPGA has 160 integer variables. Especially for design space exploration, when the optimization may be repeated several times, running a MINLP solver within an exploration loop might turn out to be prohibitive.

For this reason, we propose a heuristic formulation that separates the optimization in two steps. The first step determines the total number of CUs for each kernel to minimize II. The second step allocates the CUs to the available FPGAs.

First Step: Geometric Programming

If we disregard the spreading minimization, i.e. $\beta = 0$ in (3.5), and relax the problem by letting $n_{k,f}$ take real values, the problem becomes fully symmetric across the F identical FPGAs. This implies a symmetric solution with an equal distribution of the CUs across the F FPGAs.

Let us define $\hat{n}_k \in \mathbb{R}$ the CUs that would be equally distributed. The total number of CUs of kernel k will be

$$\hat{N}_k = F \cdot \hat{n}_k. \tag{3.14}$$

Since we want to guarantee that at least one CU is instantiated per kernel, i.e. $\hat{N}_k \geq 1$, it is possible that $\hat{n}_k = \hat{N}_k / F$ be less than one[2].

Kernel execution time and II become

$$\begin{aligned} \hat{\mathrm{ET}}_k &= \frac{\mathrm{WCET}_k}{\hat{N}_k}, \quad \forall k \in K & (3.15)\\ \hat{\mathrm{II}} &= \max_{k \in K} \hat{\mathrm{ET}}_k. & (3.16)\end{aligned}$$

We can thus reformulate the problem (3.5)-(3.10) with $\beta = 0$ as follows:

$$\begin{aligned} \text{minimize } \hat{g} &= \hat{\mathrm{II}} & (3.17)\\ \text{subject to} & & \\ \hat{\mathrm{II}} &\geq \hat{\mathrm{ET}}_k, \quad \forall k \in K & (3.18)\\ \hat{N}_k &\geq 1, \quad \forall k \in K & (3.19)\\ \textstyle\sum_{k=1}^{|K|} \frac{\hat{N}_k}{F} R_k &\leq R, & (3.20)\\ \textstyle\sum_{k=1}^{|K|} \frac{\hat{N}_k}{F} B_k &\leq B. & (3.21)\end{aligned}$$

Note that the number of unknowns $\hat{N}_k$ is F times less than the number of unknowns $n_{k,f}$ in the original formulation.

The minimization of $\hat{\mathrm{II}}$ in (3.17)-(3.21) is compatible with a Geometric Programming (GP) formulation. GP problems are solved quickly even with hundreds of variables. Therefore, we use a GP solver as the first step in our heuristic to determine $\hat{N}_k$ for all kernels.

Second Step: FPGA Allocation

Before allocation, the variables $\hat{N}_k \in \mathbb{R}$ must be discretized so as to obtain $N_k \in \mathbb{N}$. The integrality is enforced by a branch-and-bound technique similar to

[2] We can liken n_k to the *average* number of CU of kernel k across F FPGAs.

those used in ILP. Two subproblems are generated with $N_k \leq \lfloor \hat{N}_k \rfloor$ and $N_k \geq \lceil \hat{N}_k \rceil$. The search is pruned when the cost of a sub-problem is greater than the best cost found. 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 and the fact that the number of kernels is limited (e.g. around 20 for the VGG benchmark). The MINLP approach, on the other hand, must discretize every variable, and hence may potentially have a much larger branching tree.

For simplicity, from now we use the general term *resource constraint* to refer to both actual resource and bandwidth constraints.

The N_k CUs are allocated with a greedy heuristic. The rationale is to *allocate the critical kernels first*. These are the kernels for which a CU reduction has a significant impact on II, hence they should *all* be allocated. After each allocation of a kernel, either full or partial, the kernels are sorted in decreasing criticality order. Moreover, by sorting the FPGAs after each allocation in increasing order of resource slack, the heuristic tends to consolidate the kernels by allocating *all* the CUs to already occupied FPGAs while not exceeding the resource constraints. If it is not possible to allocate all of them, the heuristic allocate as many CUs as possible starting from the least occupied FPGA.

The pseudo-code of the heuristic is shown in Algorithm 1. We search for possible solutions in the vicinity of the initial resource constraint R used in the GP step. We define T as the maximum deviation from the initial constraint. We define Δ as the step by which the current resource constraint R_c, initialized as R, is updated at each iteration, i.e. $R_c = R_c + \Delta$. The iterations continue while $R_c < R + T$.

The for loop at line 11 partially allocates the CUs of kernels that cannot fit in one single FPGA, if any. The for loop at line 23 attempts to allocate all of the remaining CUs starting from the most occupied FPGA (while loop at line 26) and, if not possible, it allocates as many CUs as possible in the least occupied FPGA (lines 31-34).

3.3 Experimental Results

We implemented our allocation heuristic in C++ and linked it to an existing efficient GP solver [55]. To validate our optimization method we used two widely used CNNs, AlexNet [5] and VGG [6]. For AlexNet, we considered both 32-bit floating point and 16-bit fixed point versions, to which we refer in the following as Alex-16 and Alex-32, respectively. For VGG, we considered only the 16-bit fixed point version. We experimented with different numbers of FPGAs, from 2 to 8, and with different resource constraints.

Algorithm 1: Pseudo-code of heuristic allocation

```
procedure AllocateCUs($N_k, T, R, \Delta$)
    $\mathrm{CU} = (CU_1, CU_2, \ldots, CU_{|K|})$            // Vector of kernel CUs to allocate
    $\mathrm{CU}_k = N_k, \forall k$                          // CUs to allocate initialized to GP values
    $R_c = R$                                                 // FPGA resource constraint initialized to GP value
    $S = (S_1, S_2, \ldots, S_F)$                             // Vector of FPGA resource slack
    $S_f = R, \forall f$                                      // FPGA resource slack initialized to constraint value
    $n_{k,f} = 0, \forall k, f$                               // Allocated CUs initialized to zero
    $alloc = \mathrm{FALSE}$
    while $R_c < R + T$ and not $alloc$ do
        sortCU(CU,$K$)                                        // Sort kernels by descending criticality
        for $k = 1$ to $|K|$ do // Allocate large kernels first
            $f = 1$
            while $\mathrm{CU}_k \cdot R_k > R$ do
                if $S_f = R$ then
                    $\delta\mathrm{CU} = \lfloor R/R_k \rfloor$
                    $\mathrm{CU}_k = \mathrm{CU}_k - \delta\mathrm{CU}$
                    $S_f = S_f - \delta\mathrm{CU} \cdot R_k$
                    $n_{k,f} = n_{k,f} + \delta\mathrm{CU}$
                else
                    $f = f + 1$
        sortCU(CU,$K$)
        sortFPGA($S$)                                         // Sort FPGAs by increasing slack
        for $k = 1$ to $|K|$ do // Allocate all kernels
            $partial\_alloc = \mathrm{FALSE}$
            $f = 1$
            while $f \leq F$ and not $partial\_alloc$ do
                if $S_f \geq \mathrm{CU}_k \cdot R_k$ then
                    $S_f = S_f - CU_k \cdot R_k$
                    $n_{k,f} = n_{k,f} + \mathrm{CU}_k$
                    $\mathrm{CU}_k = 0$
                    $partial\_alloc = \mathrm{TRUE}$
                $f = f + 1$
            if $\mathrm{CU}_k > 0$ then
                // Use the space of least used FPGA ($F$), if possible $\delta\mathrm{CU} = \lfloor S_F/R_k \rfloor$
                $\mathrm{CU}_k = \mathrm{CU}_k - \delta\mathrm{CU}$
                $S_F = S_F - \delta\mathrm{CU} \cdot R_k$
                $n_{k,F} = n_{k,F} + \delta\mathrm{CU}$
            sortFPGA($S$)
        if $\sum_k \mathrm{CU}_k > 0$ then
            $R_c = R_c + \Delta$
        else
            $alloc = \mathrm{TRUE}$                           // All kernels allocated
```

Tables 3.2-3.3 show the results of the initial characterization of the various kernels of these applications when implemented on one FPGA of the AWS F1 instance[3]. For space reasons we report only DSP and BRAM resource use, especially because

[3] While the kernel code for AlexNet has been fully optimized, and performance results are in line with the literature, the VGG kernels have not yet been fully optimized. Again, our goal is to show how CUs can be allocated, not to discuss how their internal code can be massaged for HLS.

these resources are much more critical than LUTs and FFs in our experiments.

Table 3.2: Characterization of kernels for Alex-32 (AlexNet 32-bit floating point) and Alex-16 (AlexNet 16-bit fixed point).

	Alex-32				**Alex-16**			
Kernels	**BRAM (%)**	**DSP (%)**	**BW (%)**	**WCET (ms)**	**BRAM (%)**	**DSP (%)**	**BW (%)**	**WCET (ms)**
CONV1	13.07	21.24	1.3	13	10.59	4.31	1.8	5.16
POOL1	2.84	0	7.03	1.78	0.05	0	3.5	1.78
NORM1	6.1	2.11	5.7	0.839	2.53	0.06	3.1	0.78
CONV2	8.73	37.59	2.4	7.19	4.39	7.63	2.1	4.11
NORM2	7.75	2.11	3.7	0.807	6.66	0.06	2.2	0.67
CONV3	5.22	28.13	5.0	7.78	2.63	5.66	2.9	6.7
CONV4	2.13	37.5	3.7	9.08	1.91	7.55	3.2	5.06
CONV5	8.73	37.5	4.2	4.84	4.39	7.55	3.1	3.29
SUM	54.57	166.18	33.1	45.32	33.15	32.82	21.9	27.55

Table 3.3: Characterization of VGG kernels (16-bit fixed point).

Kernels	**BRAM (%)**	**DSP (%)**	**BW (%)**	**WCET (ms)**
CONV1	3.67	2.95	2.0	28.8
CONV2	9.97	15.14	2.1	67.8
POOL2	11.62	0.03	5.2	13.3
CONV3	9.97	15.14	2.3	22.7
CONV4	9.97	15.14	2.4	32.1
POOL4	2.94	0.03	5.1	6.9
CONV5	8.32	15.07	2.0	22.8
CONV6, 7	8.32	15.05	2.3	32.9
POOL7	1.5	0.03	5.0	3.5
CONV8	2.12	15.02	2.1	24.5
CONV9, 10	2.12	15.02	2.5	37.7
POOL10	0.05	0.01	4.0	2.1
CONV11,12,13	2.12	14.99	2.6	20.3
SUM	87.37	183.67	49.7	0.4 (s)

Before reporting the details of the comparison of our heuristic with a state-of-the-art MINLP solver [2], we report on the evaluation of the effect of changing the T parameter of the heuristic while keeping the other parameter Δ set to 1%. We report the result of this analysis for Alex-16 in Figure 3.2. Similar results are

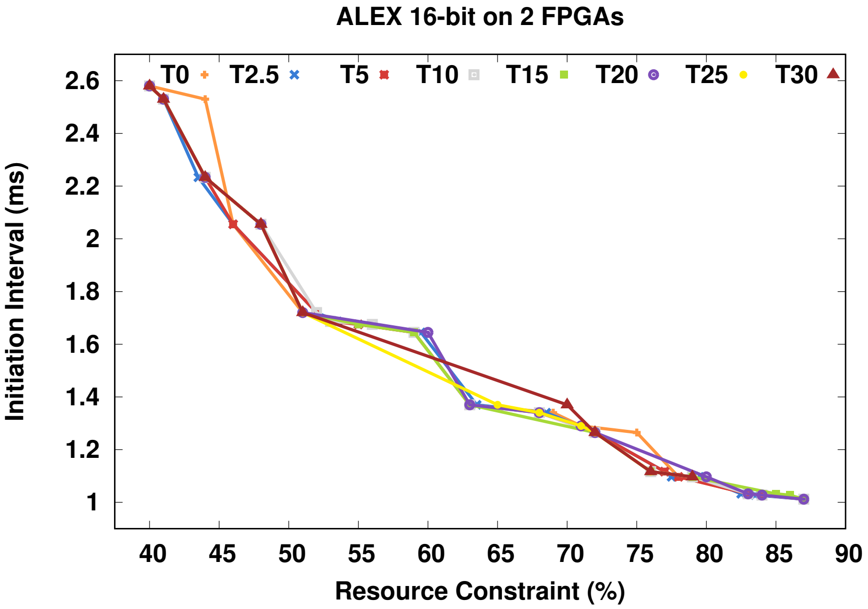


Figure 3.2: Alex-16 results with different values of T (in %).

obtained for Alex-32 and VGG. We observe little effect of T on the value of II across a large range of resource constraints. Therefore, the following results have all been obtained with T=0%.

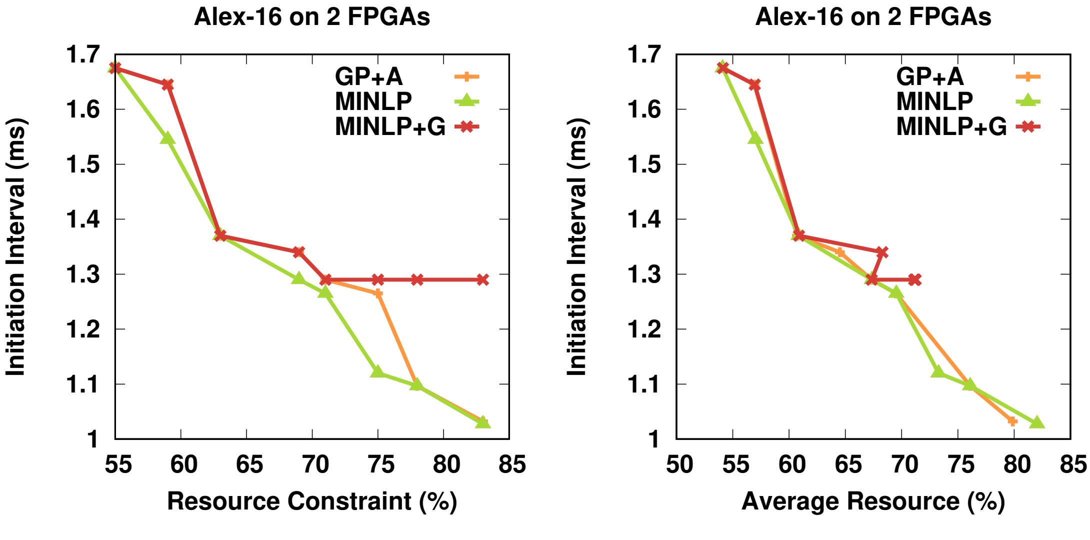
We ran all our optimization algorithms on a multi-core CPU (Intel Core i7-2600 @3.40 GHz, 4 Cores, 8 Threads) with 16-GB DDR3 DRAM @1333 MHz from Micron and with Linux CentOS (release 6.10), we are using one processor for our experiments, and our FPGA accelerations on AWS F1 instances with 8 FPGAs.

Out of all our experiments we selected three representative cases of the spectrum of possible multi-FPGA implementations: Alex-16 on 2 FPGAs, Alex-32 on 4 FPGAs, and VGG on 8 FPGAs. For these three cases, Table 3.4 shows the value of the two weights α and β. These values are chosen in such a way to equalize the relative importance of II and ϕ in the optimization function g in (3.5).

Table 3.4: Parameters for the spreading function

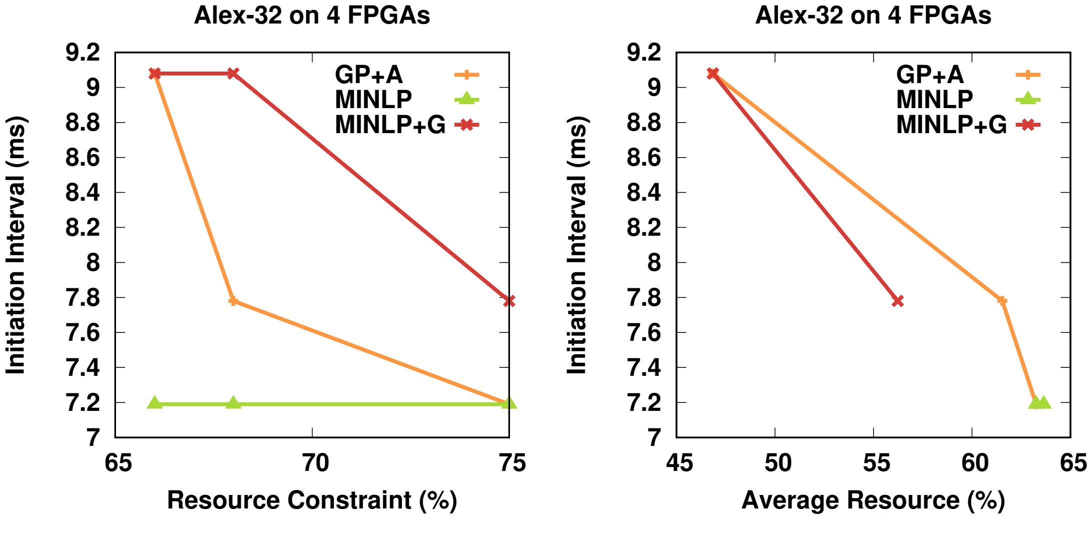
Applications	α	β
Alex-16 on 2 FPGAs	1	0.7
Alex-32 on 4 FPGAs	1	6
VGG on 8 FPGAs	1	50

The left graphs in Figs. 3.3-3.5 report the results of II obtained by changing



(a) (b)

Figure 3.3: AlexNet 16-bit fixed-point on 2 FPGAs.



(a) (b)

Figure 3.4: AlexNet 32-bit floating-point on 4 FPGAs.

the resource constraint, i.e. the maximum allowed FPGA resource utilization. (Incidentally, the most critical resources in all our experiments are DSPs.) The right graphs show the same points of the left graphs in a different space of II versus average FPGA resource utilization. The labels in the figure keys are as follows:

- **GP+A** refers to the heuristic consisting of GP (optimizing II) and allocation (discretizing and optimizing spreading);

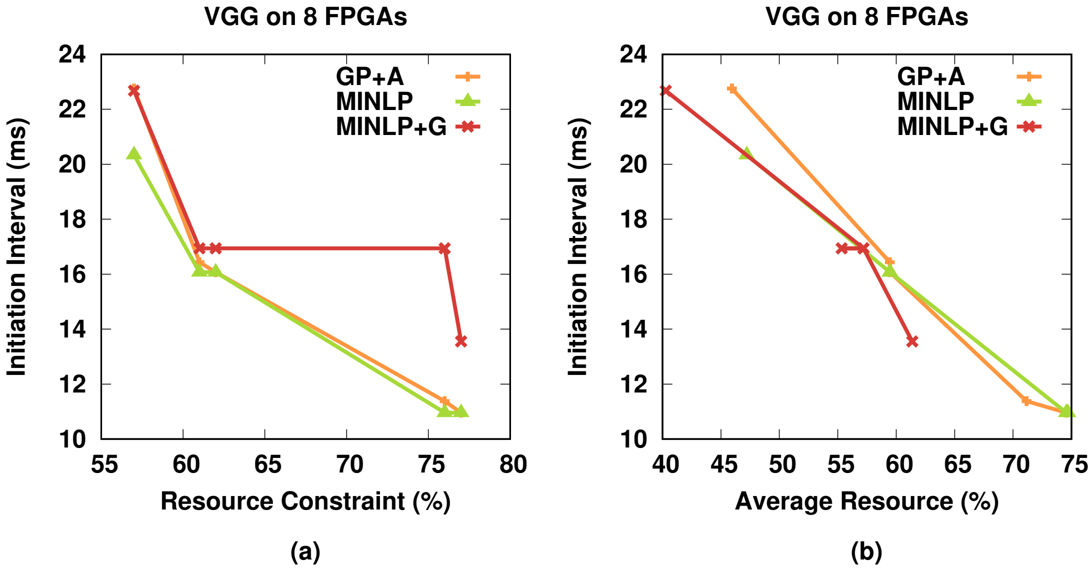


Figure 3.5: VGG 16-bit fixed-point on 8 FPGAs.

- **MINLP** refers to the MINLP solver set up to optimize only II and not the spreading (i.e. $\beta = 0$)[4];
- **MINLP+G** refers to the MINLP solver set up to optimize both II and spreading (i.e. α and β as in Table 3.4).

As expected, the left graphs show that MINLP obtains the best II for a given resource constraint when the spreading is ignored. With the exception of Alex-16 at low resource utilization, GP+A tracks well MINLP and in particular it catches the extremes. The results on the right graphs show that II nicely scales down as the average resource increases, especially for MINLP and GP+A.

The Alex-16 case is relevant because it shows that in some cases especially in the lower range of resource constraint, GP+A cannot reach the same performance of MINLP, but indeed behaves more similarly to MINLP+G. This is because both GP+A and MINLP+G tend to consolidate the CUs in fewer FPGAs than what MINLP does. This might result in a performance loss—25% in Figure 3.4(a) at the lowest resource constraint—but in a better average FPGA utilization: Figure 3.4(b) shows around 40% less average utilization of GP+A and MINLP+G compared to MINLP at the lowest resource constraint[5].

[4] These results show the best achievable II for a given resource constraint, but they would require *an extremely complex routing of data from each CU in one layer to several other CUs spread over multiple FPGAs, each with its own DRAM banks*, and thus they would make the host code essentially unmanageable.

[5] The three MINLP points in Figure 3.4(a) represent actually the same solution, because the

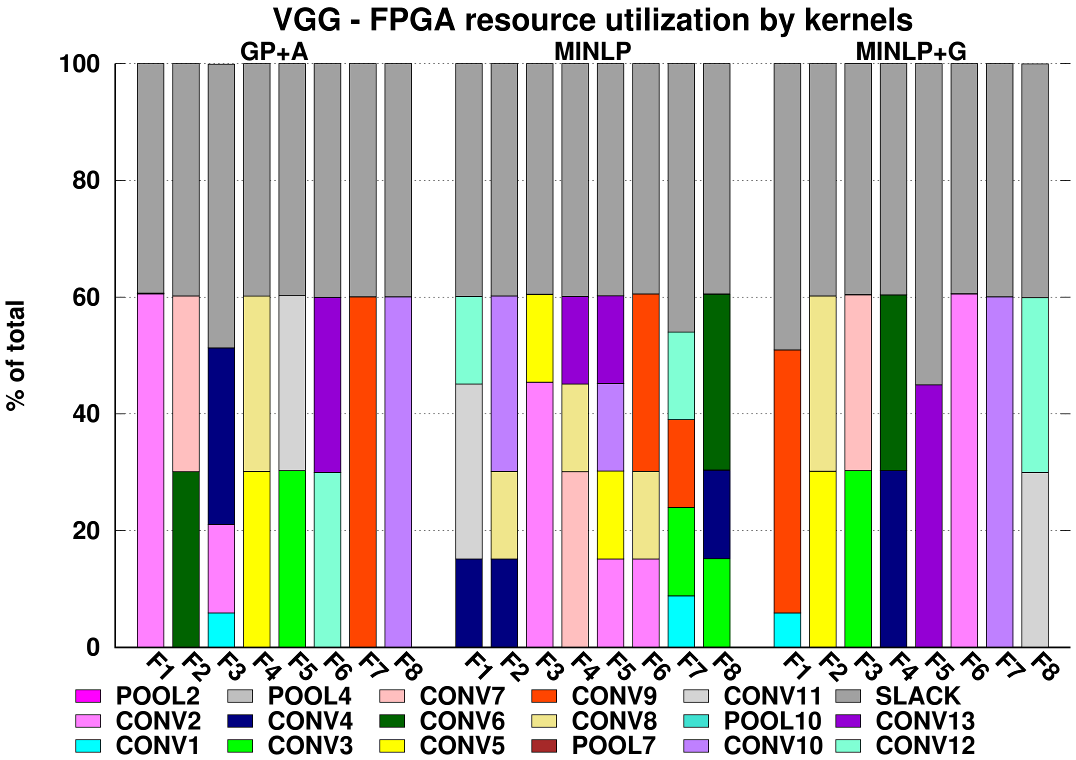


Figure 3.6: VGG resource usage for 61% resource constraint.

We report only one example of resource distribution in Figure 3.6, which refers to the VGG case with a specific resource constraint of 61%. The histograms show how the kernels are distributed across 8 FPGAs and how many resources each kernel uses while respecting the 61% resource constraint (SLACK ≥39% in figure). As expected from the previous discussion, both GP+A and MINLP+G tend to concentrate the kernels in one FPGA, whereas MINLP spreads them across multiple FPGAs.

Finally, the CPU time of GP+A ranges between 0.78 s (Alex-16 on 2 FPGAs) to 4.4 s (VGG on 8 FPGAs), whereas that of MINLP and MINLP+G ranges from around one minute to several hours, with a speedup that ranges from around 100x to around 1000x. The quality of the results and the low CPU time clearly show that our heuristic approach is suitable for design space exploration of multi-kernel applications deployed on multi-FPGA boards.

solver is able to reach the minimum II without saturating the resource utilization in any FPGA. This is more evident in Figure 3.4(b), where the three points overlap.

3.4 Conclusions

We have proposed and experimentally evaluated a new and fast method for minimizing the initiation interval of pipelined applications consisting of multiple kernels and deployed on multiple FPGAs. We optimize the number of parallel compute units (CUs) for each kernel while respecting resource and memory bandwidth constraints. The optimization problem is non-linear and with both integer (i.e. the CUs) and real variables, for which accurate MINLP solvers can be used but at the cost of long execution time. We use a two-step heuristic that first relaxes the problem by letting integer variables take real values, which allows us to use a fast geometric programming solver. Second, we discretize the results and apply a greedy allocation of the CUs over the target FPGAs, aimed at minimizing the spreading of a kernel over FPGAs. We obtain results that are comparable to what a MINLP solver can obtain, but our algorithm is 2-3 orders of magnitude faster.

Chapter 4

Enhanced Performance Optimization Model for Multi-kernel Applications on Multi-FPGA Platform

In Chapter 3 we proposed a model to optimize the performance of the multi-kernel applications like CNNs. What is missing in the model is that it does not include the data transfer time between FPGA and host CPU, as well as the frequency reduction due to the routing congestion introduced by the increasing resource utilization. In this chapter, we propose a more precise model. For the given resources, the model will return 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. 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.

This work content in this chapter was published in [56].

4.1 Introduction

Similar to the model proposed in Chpter 3, we use the same multi-FPGA platform AWS F1 instance. As shown in Figure 3.1, 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. Figure 4.1 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 Figure 4.1, each pipeline stage is mapped to a specific number of CUs ($N_1, N_2, \ldots, N_k$). 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 [57]. 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 Figure 4.1.

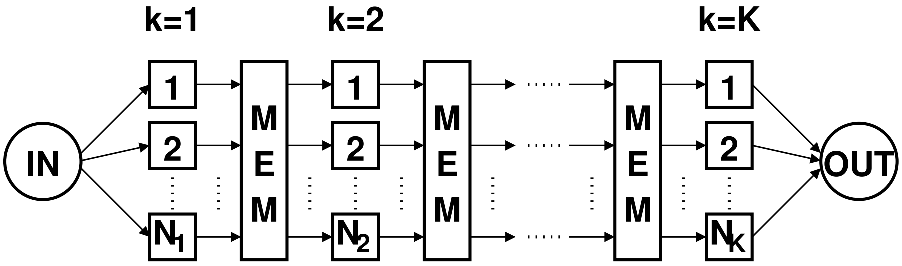


Figure 4.1: Example of a **K**-stage kernel pipeline.

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 datacenter 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:

[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.

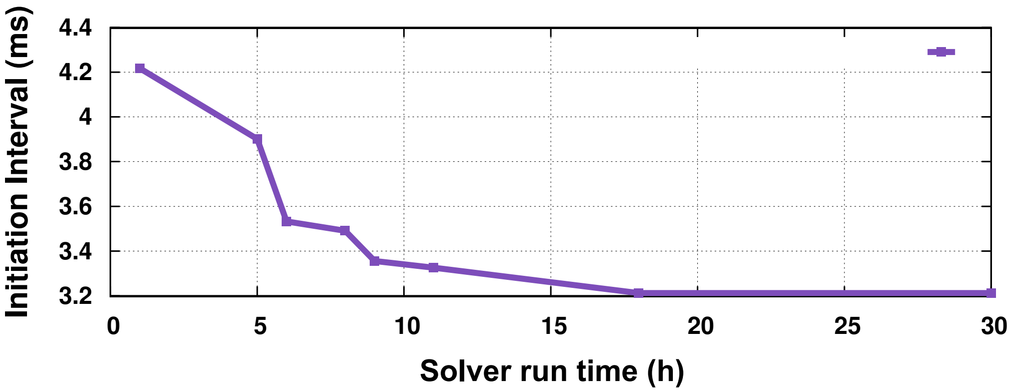


Figure 4.2: Slow progress of the MINLP solver while searching for the optimum allocation of a CNN application.

- 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, Figure 4.2 shows the slow progress of the Couenne solver [2] when optimizing the Initiation Interval (II), which is the inverse of the pipeline throughput, of the YOLO CNN [58] 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 chapter 3 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 [59], VGG [6], YOLO [58] and ResNet [7]. 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.

This chapter is organized as follows. In Section 4.2, we present the problem formulation, and discuss the proposed heuristic method in Section 4.3. In Section 4.4, we present and discuss the experimental results. In Section 4.5, we compare the result obtained using the "Simplified-model" and the "Enhanced-model". Section 4.6 concludes the chapter and outlines opportunities for future work.

4.2 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 Figure 4.1. However, unlike [23, 26, 27, 28] 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

$$N_k = \sum_{\mathbf{f}=1}^{\mathsf{F}} n_{k,f}, \tag{4.1}$$

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 external DRAM the intermediate data and constants needed for their computation through AXI ports as shown in Figure 3.1. 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.

[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.

We formulate the optimization problem as follows:

$$\begin{aligned} &\text{minimize} \quad II && (4.2)\\ &\text{subject to} &&\\ N_k &\geq 1, \quad \forall k && (4.3)\\ \sum_k n_{k,f}\mathbf{R}_{k,t} &\leq \mathbf{R}_t, \quad \forall f,\ \forall t, && (4.4) \end{aligned}$$

where the goal is to minimize II. Constraint (4.3) guarantees that each kernel is implemented with at least one CU. Constraint (4.4) 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 4.1 and Table 4.2, respectively. Note that we use bold typefaces for constants and regular typefaces for variables.

4.2.1 Modeling of Application Initiation Interval (II)

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

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

$$II = T_{h2f} + T_{exe} + T_{f2h}. \tag{4.5}$$

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.

Table 4.1: Constants (boldface) used in model equations.

Notation	Description
$\mathbf{K}$	number of kernels: $k \in \{1, \ldots, \mathbf{K}\}$ used as index
$\mathbf{F}$	number of FPGAs: $f \in \{1, \ldots, \mathbf{F}\}$ 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
γ_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

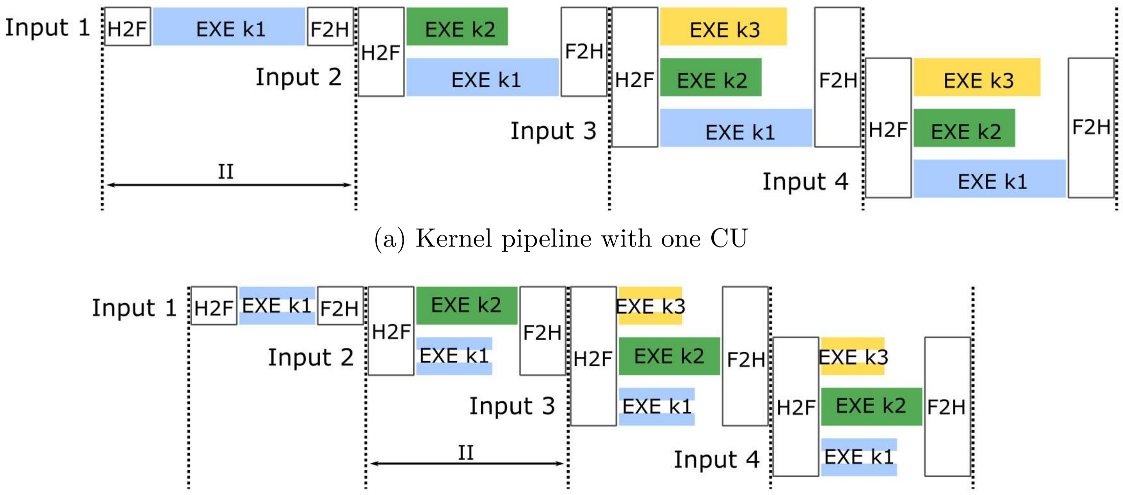
Figure 4.3 shows an example of pipelined execution of three kernels. In Figure 4.3(a), each kernel is implemented with one CU. In Figure 4.3(b), kernels $K1$ and $K3$ use two CUs each, which leads to a significantly lower II. Note how the duration of the EXE phase is related to the maximum execution time among the various kernels. The kernels that determine this maximum might change, depending on the number of CUs: in Figure 4.3(a), $K1$ sets the II, while in Figure 4.3(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.

Table 4.2: 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 $\widehat{N_k}$



(a) Kernel pipeline with one CU

(b) Kernel pipeline with multiple CUs

Figure 4.3: Initiation Interval (II) depends on the number of compute units of each kernel in a multi-kernel pipeline.

4.2.2 Host-to-FPGA (H2F) Phase

The duration of this phase is

$$T_{h2f} = \frac{DI_{H2F}}{\mathbf{B}_{H2F}}, \tag{4.6}$$

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 Figure 3.1). 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 ($a_k = 0$):

$$a_k = \bigvee_{k>1,\forall f} \left((n_{k-1,f} = N_{k-1}) \wedge (n_{k,f} = N_k)\right). \tag{4.7}$$

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 (4.7) 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

$$DI_D = \sum_{k=1}^{\mathbf{K}} a_k \mathbf{DI}_k. \tag{4.8}$$

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 [7] violate the linear pipeline scheme of Figure 4.2 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 (4.8); 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:

$$\alpha_{k,f} = \begin{cases} 1 & \text{if} \quad n_{k,f} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

$$\alpha_k = \sum_{f=1}^{\mathbf{F}} \alpha_{k,f}. \tag{4.10}$$

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

$$DI_{H2F} = \sum_{k=1}^{\mathbf{K}} \alpha_k (1 - a_k) \mathbf{DI}_k. \tag{4.11}$$

Figure 4.4 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 $K2$.

4.2.3 FPGA-to-Host (F2H) Phase

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

$$T_{f2h} = \frac{DO_{F2H}}{\mathbf{B}_{F2H}}, \tag{4.12}$$

[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.

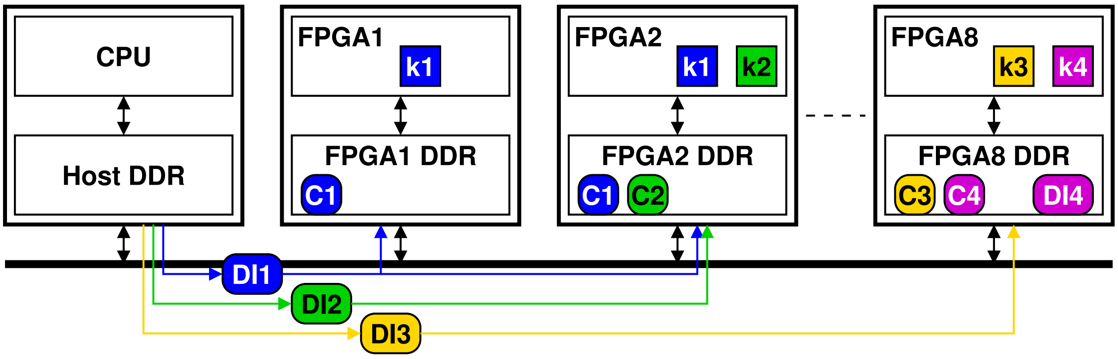


Figure 4.4: Host-to-FPGA (H2F) example showing data transfer between host DDR and FPGA DDRs. $\mathbf{DI}_4$ is not transferred because kernels $K3$ and $K4$ are on the same FPGA. $\mathbf{DI}_2$ is transferred because parts of $K1$ are on different FPGA than $K2$.

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:

$$b_k = \bigvee_{k<\mathbf{K},\forall f} ((n_{k,f} = N_k) \wedge (n_{k+1,f} = N_{k+1})) \,. \tag{4.13}$$

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

$$DO_D = \sum_{k=1}^{\mathbf{K}} b_k \mathbf{DO}_k \tag{4.14}$$

$$DO_{F2H} = \sum_{k=1}^{\mathbf{K}} (1-b_k)\mathbf{DO}_k. \tag{4.15}$$

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.

Figure 4.5 shows the F2H phase of the same hypothetical allocation in Figure 4.4. 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

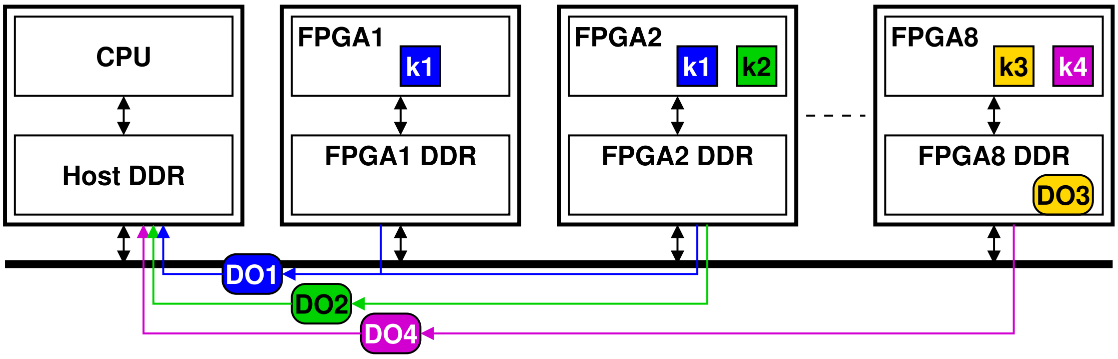


Figure 4.5: FPGA-to-Host (F2H) example showing data transfer between FPGA DDRs and host DDR. $K3$ and $K4$ are on the same FPGA and $\mathbf{DO}_3$ does not need to be transferred.

to group the CUs on a single FPGA. And for the same reason, consecutive kernels are also grouped together whenever possible.

4.2.4 Processing (EXE) Phase

In comparison with our previous model in chapter 3, 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]:

$$ET_{k,f} = TR_{k,f} + TC_{k,f} + TW_{k,f}. \tag{4.16}$$

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

$$T_{exe} = \max_{k,\,f} ET_{k,f} \tag{4.17}$$

The three times in (4.16) depend on the number of CUs of kernel k, N_k, as shown in the following.

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

[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.

data, 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:

$$dr_k = \frac{\boldsymbol{\delta}_k \mathsf{DI}_k + \boldsymbol{\gamma}_k \mathsf{C}_k}{N_k} + (1 - \boldsymbol{\delta}_k)\mathsf{DI}_k + (1 - \boldsymbol{\gamma}_k)\mathsf{C}_k, \tag{4.18}$$

where $0 \leq \boldsymbol{\delta}_k \leq 1$ and $0 \leq \boldsymbol{\gamma}_k \leq 1$. (4.18) 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 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 the total AXI bandwidth used by the CU and the portion of DDR bandwidth that the CU receives:

$$BR_{k,f} = \mathsf{x}_k \cdot \min\left(BX_f, \frac{\mathsf{BDR}}{NR_f}\right), \tag{4.19}$$

where 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 r_k and rw_k, respectively. Therefore, we have

$$\mathsf{x}_k = \mathsf{r}_k + \mathsf{rw}_k. \tag{4.20}$$

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 (4.19), and consequently the *approximated reading time* becomes

$$TR_{k,f} \simeq \frac{dr_k}{BR_{k,f}}. \tag{4.21}$$

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:

$$dw_k \quad = \quad \frac{\mathbf{DO}_k}{N_k}. \tag{4.22}$$

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

$$\mathbf{y}_k = \mathbf{w}_k + \mathbf{rw}_k. \tag{4.23}$$

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

$$BW_{k,f} = \mathbf{y}_k \cdot \min\left(BX_f, \frac{\mathbf{BDW}}{NW_f}\right). \tag{4.24}$$

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 (4.24), we obtain the *approximated writing time*:

$$TW_{k,f} \simeq \frac{dw_k}{BW_{k,f}}. \tag{4.25}$$

Computing

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

$$L_k = \frac{\mathsf{\mathbf{L1}}_k}{N_k}. \tag{4.26}$$

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:

$$F_{k,f} = \mathsf{\mathbf{F1}}_k - \psi \cdot R_f, \tag{4.27}$$

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

$$F_f = \min_k F_{k,f} \tag{4.28}$$

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

$$TC_{k,f} = \frac{L_k}{F_f}. \tag{4.29}$$

4.3 Geometric Programming and Allocator

Similar to chapter 3, MINLP solver can lead to a long optimization time. For this reason, we propose a heuristic formulation that separates the optimization in two steps. The first step determines 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**.

[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.

4.3.1 Geometric Programming

To use GP [4], 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

$$\widehat{N}_k = \mathsf{F} \cdot \widehat{n}_k. \tag{4.30}$$

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

$$\widehat{L}_k = \frac{\mathbf{L1}_k}{\widehat{N}_k}, \quad \forall k \in K \tag{4.31}$$

and the kernel computing time becomes

$$\widehat{TC}_k = \frac{\widehat{L}_k}{\mathbf{F1}_k}, \quad \forall k \in K, \tag{4.32}$$

where we use $\mathbf{F1}_k$ 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 4.4, 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 (4.2)–(4.4) as follows:

$$\begin{aligned} \text{minimize} \quad & \widehat{TC} && (4.33)\\ \text{subject to} \quad & && \\ \widehat{TC} \quad & \geq \quad \widehat{TC}_k, \quad \forall k \in K && (4.34)\\ \widehat{N}_k \quad & \geq \quad 1, \qquad \forall k \in K && (4.35)\\ \textstyle\sum_k \frac{\widehat{N}_k}{\mathsf{F}} \mathbf{R}_{k,t} \quad & \leq \quad \mathbf{R}_t. && (4.36) \end{aligned}$$

The new formulation in (4.33)–(4.36) is compatible with GP requirements [4], and as such can be solved very efficiently. Once we obtain the (fractional) number of

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

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.

4.3.2 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 search space.

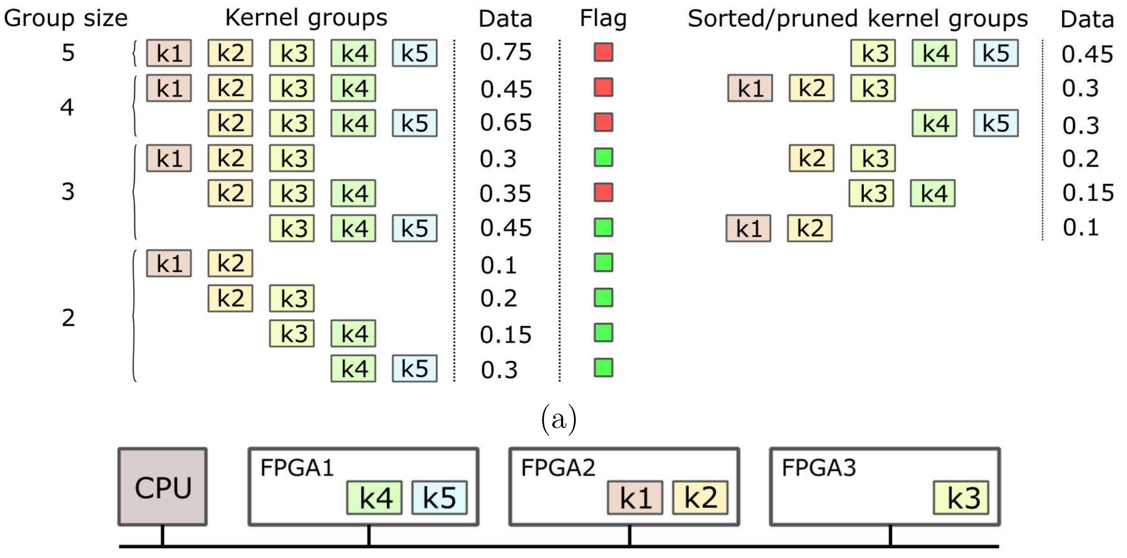
For each sub-problem generated with the discretization, we perform the actual allocation, which consists of two phases:

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

Kernel Group Allocation

A repetitive pattern can be observed from the results generated using MINLP solver, the solver tries to group the kernel groups which has a huge amount of data transfer on a single FPGA, thus to minimize the H2F and F2H transfer times, In the heuristic kernel allocation, we also 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 Figure 4.6(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 Figure 4.6(a).



(a)

(b)

Figure 4.6: 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.

Based on this list and on FPGA resource constraints, we allocate the groups using the greedy heuristic procedure called *AllocateGroups* in Algorithm 2.

After pruning and sorting the groups (N_g is the set of groups and its cardinality is $|N\_g| = \sum_{n=1}^{\mathbf{K}-1} n$), the loops in lines 9–21 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. Figure 4.6(b) shows

Algorithm 2: Pseudo-code of kernel group allocation

```
procedure AllocateGroups($N_k, R$)
    CU = $(CU_1, CU_2, \dots, CU_{\mathsf{K}})$        // Kernel CUs to allocate
    $\mathrm{CU}_k = N_k, \forall k$        // Initialized to GP values
    $S = (S_1, S_2, \dots, S_{\mathsf{F}})$        // FPGA resource slacks
    $S_f = R, \forall f$        // Initialized to resource constraint
    $n_{k,f} = 0, \forall k, f$        // Allocated CUs initialized to zero
    $N_g$ = All_Kernel_groups        // Set of all possible groups
    $N_g = N_g \setminus$ Infeasible_groups($N_g$)        // Pruning
    $N_g$ = Sorted_groups($N_g$)        // Sorting by data size
    $R_g$ = Group_resources($N_g$)        // Resources needed by
                                          // each group
    for $n = 1$ to $|N\_g|$ do        // Try to allocate group $n$

        for $f = 1$ to $\mathsf{F}$ do
            if $R\_g[n] \leq S_f$ then        // if $f$ has space left

                $S_f = S_f - R\_g[n]$
                for $k \in N\_g[n]$ do        // all kernels in $n$

                    $n_{k,f} = \mathrm{CU}_k$
                    $\mathrm{CU}_k = 0$
                $N_g = N_g \setminus \{N_g[n]\}$        // remove group $n$
                sortFPGA($S$)        // Sort by increasing slack

    return CU, $S$
```

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 the following.

Individual Kernel Allocation

Before delving into the details of the procedure shown in Algorithm 3, 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 2–34 of Algorithm 4, with the boundary increased on line 32 and the exit condition (in case of allocation) on line 34.

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 4–13 and 15–30, 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 6) to still *empty* FPGAs (line 7).

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 25), we use the least used FPGA, which is the last in the ordered set ($\mathbf{F}$, 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 31), 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

Algorithm 3: Pseudo-code of kernel allocation

```
procedure AllocateKernels($\mathrm{CU}, S$)
    $R_c = R$                                                   // Resource constraint initialized
    $alloc = \mathrm{FALSE}$
    while $R_c < R + T$ and not $alloc$ do
        sortKernels(CU)                                         // Sort by descending criticality
        for $k = 1$ to $\boldsymbol{K}$ do // Allocate large kernels first
            $f = 1$
            while $\mathrm{CU}_k \cdot R_k > R$ do // Can't fit in one FPGA
                if $S_f = R$ then
                    $\delta\mathrm{CU} = \lfloor R/R_k \rfloor$
                    $\mathrm{CU}_k = \mathrm{CU}_k - \delta\mathrm{CU}$
                    $S_f = S_f - \delta\mathrm{CU} \cdot R_k$
                    $n_{k,f} = n_{k,f} + \delta\mathrm{CU}$
                else
                    $f = f + 1$
        sortFPGA($S$)                                           // Sort by ascending slack
        for $k = 1$ to $\boldsymbol{K}$ do // Allocate all kernels
            $partial\_alloc = \mathrm{FALSE}$
            $f = 1$
            while $f \leq \mathsf{F}$ and not $partial\_alloc$ do
                if $S_f \geq \mathrm{CU}_k \cdot R_k$ then
                    $S_f = S_f - \mathrm{CU}_k \cdot R_k$
                    $n_{k,f} = n_{k,f} + \mathrm{CU}_k$
                    $\mathrm{CU}_k = 0$
                    $partial\_alloc = \mathrm{TRUE}$
                $f = f + 1$
            if not $partial\_alloc$ then
                // Use least used FPGA F, if possible
                $\delta\mathrm{CU} = \lfloor S_{\mathsf{F}}/R_k \rfloor$
                $\mathrm{CU}_k = \mathrm{CU}_k - \delta\mathrm{CU}$
                $S_{\mathsf{F}} = S_{\mathsf{F}} - \delta\mathrm{CU} \cdot R_k$
                $n_{k,\mathsf{F}} = n_{k,\mathsf{F}} + \delta\mathrm{CU}$
            sortFPGA($S$)
        if $\sum_k \mathrm{CU}_k > 0$ then // Not all CUs are allocated
            $R_c = R_c + \Delta$
        else
            $alloc = \mathrm{TRUE}$                             // All kernels allocated
    if $alloc$ then // All CUs are allocated
        return $n_{k,f}, \forall k, \forall f$
    else
        return allocation failed
```

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.

4.4 Experimental Results

We implemented our allocation heuristics in C++ and linked it to an existing GP solver [55]. To validate our optimization method, we use several widely used CNNs: AlexNet [59], VGG-net [6], YOLO [58] and ResNet [7]. 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. We are using one processor for our experiments. Our hardware platform is 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 4.3–4.6, 4.8 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 time. 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

Table 4.3: Characterization of kernels for Alex-32 (floating point). C, P, N stand for convolutional, pooling and normalization layers.

Kernels	$\mathbf{DI_k}$ (MB)	$\mathbf{DO_k}$ (MB)	$\mathbf{C_k}$ (MB)	δ_k	γ_k	$\mathbf{rw_k}$	$\mathbf{F1_k}$ (GHz)	DSP (%)	$\mathbf{TC1_k}$ (ms)
C1	0.62	1	0	0	1	1	0.25	21.24	4.41
P1	1	0.27	0	1	1	1	0.25	0	0.11
N1	0.27	0.27	0	1	1	1	0.25	2.11	0.29
C2	0.27	0.17	1.17	0	1	1	0.22	37.59	2.99
N2	0.17	0.17	0	1	1	1	0.223	7.75	0.2
C3	0.17	0.25	3.375	0	1	1	0.214	28.13	2.18
C4	0.25	0.25	2.53	0	1	1	0.21	37.5	1.82
C5	0.25	0.035	1.69	0	1	1	0.22	37.5	3.73

Table 4.4: Characterization of kernels for Alex-16 (fixed-point). C, P, N stand for convolutional, pooling and normalization layers.

Kernels	$\mathbf{DI_k}$ (MB)	$\mathbf{DO_k}$ (MB)	$\mathbf{C_k}$ (MB)	δ_k	γ_k	$\mathbf{rw_k}$	$\mathbf{F1_k}$ (GHz)	DSP (%)	$\mathbf{TC1_k}$ (ms)
C1	0.31	0.58	0	0	1	1	0.25	4.31	2.63
P1	0.58	0.139	0	1	1	1	0.25	0.58	0.37
N1	0.139	0.139	0	1	1	1	0.25	0.06	0.28
C2	0.139	0.086	0.614	0	1	1	0.25	7.63	1.927
N2	0.086	0.086	0	1	1	1	0.25	0.06	0.17
C3	0.086	0.13	1.77	0	1	1	0.25	5.66	1.82
C4	0.13	0.13	1.33	0	1	1	0.25	7.55	1.08
C5	0.13	0.018	0.884	0	1	1	0.25	7.55	1.72

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 4.9, 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 (4.27).

Table 4.5: Characterization of kernels (K) for YOLO-32 (floating point). C and P stand for convolutional and pooling layers.

K	$\mathbf{DI_k}$ (MB)	$\mathbf{DO_k}$ (MB)	$\mathbf{C_k}$ (MB)	δ_k	γ_k	$\mathbf{rw_k}$	$\mathbf{F1_k}$ (GHz)	DSP (%)	$\mathbf{TC1_k}$ (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 4.9: 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 4.10: Execution time of our heuristic method GP+A to generate the Pareto points in Figure 4.10.

	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

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 Figure 4.2, we decided to stop the solver. The time limit, as shown in Table 4.7, varies from 10 to 70 hours for the different cases, due to the different size of the problem.

Table 4.6: Characterization of kernels (K) for VGG-16 (fixed-point). C and P stand for convolutional and pooling layers.

K	$\mathbf{DI_k}$ (MB)	$\mathbf{DO_k}$ (MB)	$\mathbf{C_k}$ (MB)	δ_k	γ_k	$\mathbf{rw_k}$	$\mathbf{F1_k}$ (GHz)	DSP (%)	$\mathbf{TC1_k}$ (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 4.7: Time limit used by the Couenne MINLP solver to obtain one point on the II vs. R curve of each implementation in Figure 4.10.

	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 4.10 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 4, our heuristic requires to set a resource usage threshold, T. Figure 4.7 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 $T = 1\%$.

Table 4.8: Characterization of kernels (K) for ResNet, C and P stand for convolutional and pooling layers.

	$\mathbf{DI_k}$	$\mathbf{DO_k}$	$\mathbf{C_k}$	δ_k	γ_k	$\mathbf{rw_k}$	$\mathbf{F1_k}$	DSP	$\mathbf{TC1_k}$
K	(MB)	(MB)	(MB)				(GHz)	(%)	(ms)
C1	0.287	1.53	0.018	1	0	1	0.25	0.34	4.07
P1	1.53	0.383	0	1	1	1	0.25	0	0.13
C2,3,5,7	0.383	0.383	0.07	0	1	1	0.25	1.99	4.02
C4,6	0.766	0.383	0.07	0	1	1	0.25	2	4.02
C8	0.766	0.191	0.141	0	1	1	0.25	3.85	1.11
C9,11,13,15	0.191	0.191	0.281	0	1	1	0.25	3.85	2.01
C10,12,14	0.383	0.191	0.281	0	1	1	0.25	3.85	2.01
C16	0.383	0.096	0.563	0	1	1	0.25	7.58	0.56
C17,19,21,23,25,27	0.096	0.096	1.125	0	1	1	0.25	7.55	1.01
C18,20,22,24,26	0.191	0.096	1.125	0	1	1	0.25	7.55	1.01
C28	0.191	0.048	2.25	0	1	1	0.25	7.56	0.52
C29,31,33	0.049	0.048	4.5	0	1	1	0.25	7.58	0.99
C30,32	0.096	0.048	4.5	0	1	1	0.25	7.58	0.99
C34	0.383	0.048	0.008	0	1	1	0.25	3.79	0.3
C35	0.191	0.024	0.031	0	1	1	0.25	3.79	0.3
C36	0.096	0.012	0.125	0	1	1	0.25	3.79	0.3

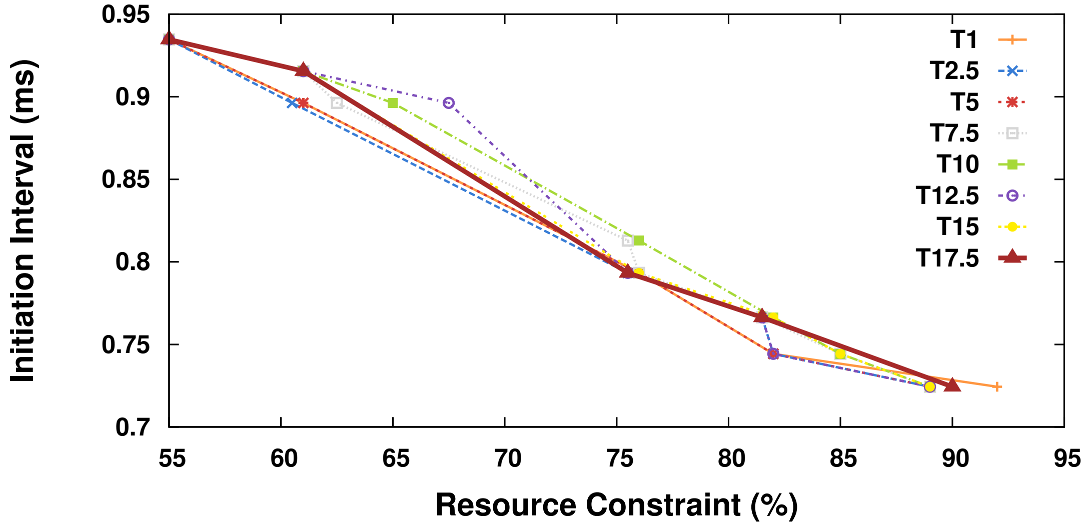


Figure 4.7: II vs. $R_{\max}$ with different resource usage thresholds for AlexNet fixed-point (Alex-16) on two FPGAs.

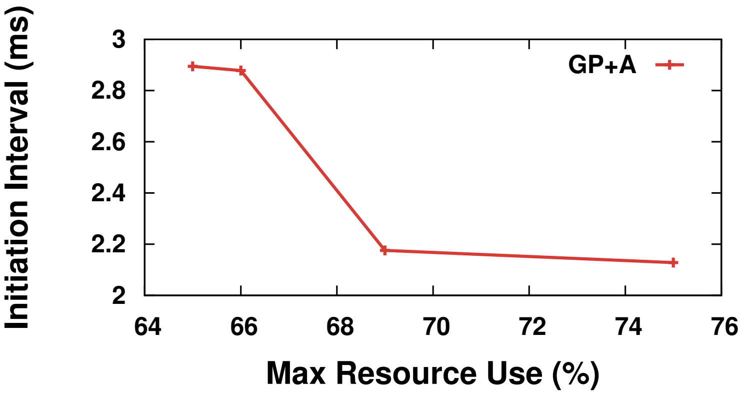


Figure 4.8: RESNET-16 on 5 FPGAs: II vs resource usage.

The plots in Figure 4.10 show the results obtained by changing the resource constraint for both the MINLP solver and our heuristic. ALEX-16 on two FPGAs, shown in Figure 4.10(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 Figures. 4.10(b)–(e), GP+A significantly outperforms MINLP (even for runs within the fairly large time limits of Table 4.7), with only one exception: the point at $R = 61\%$ for ALEX-32 on four FPGAs in Figure 4.10(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 Figure 4.8, 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 Figure 4.8 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.

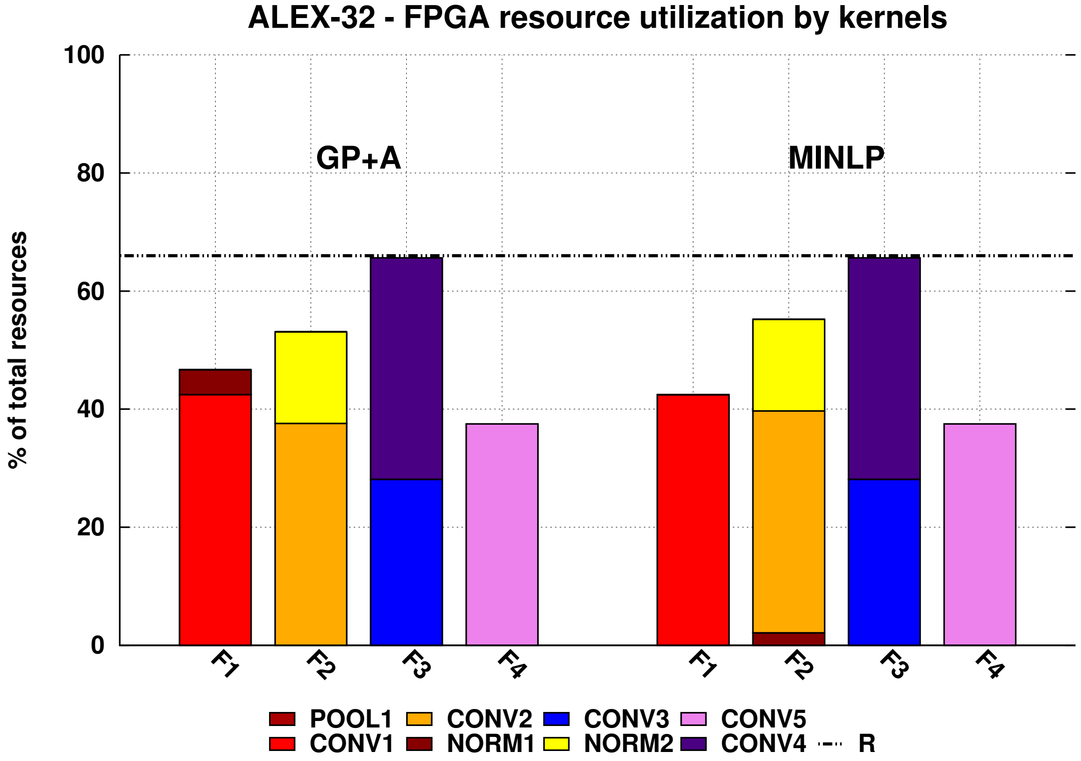


Figure 4.9: ALEX-32 allocation on 4 FPGA using GP+A and MINLP.

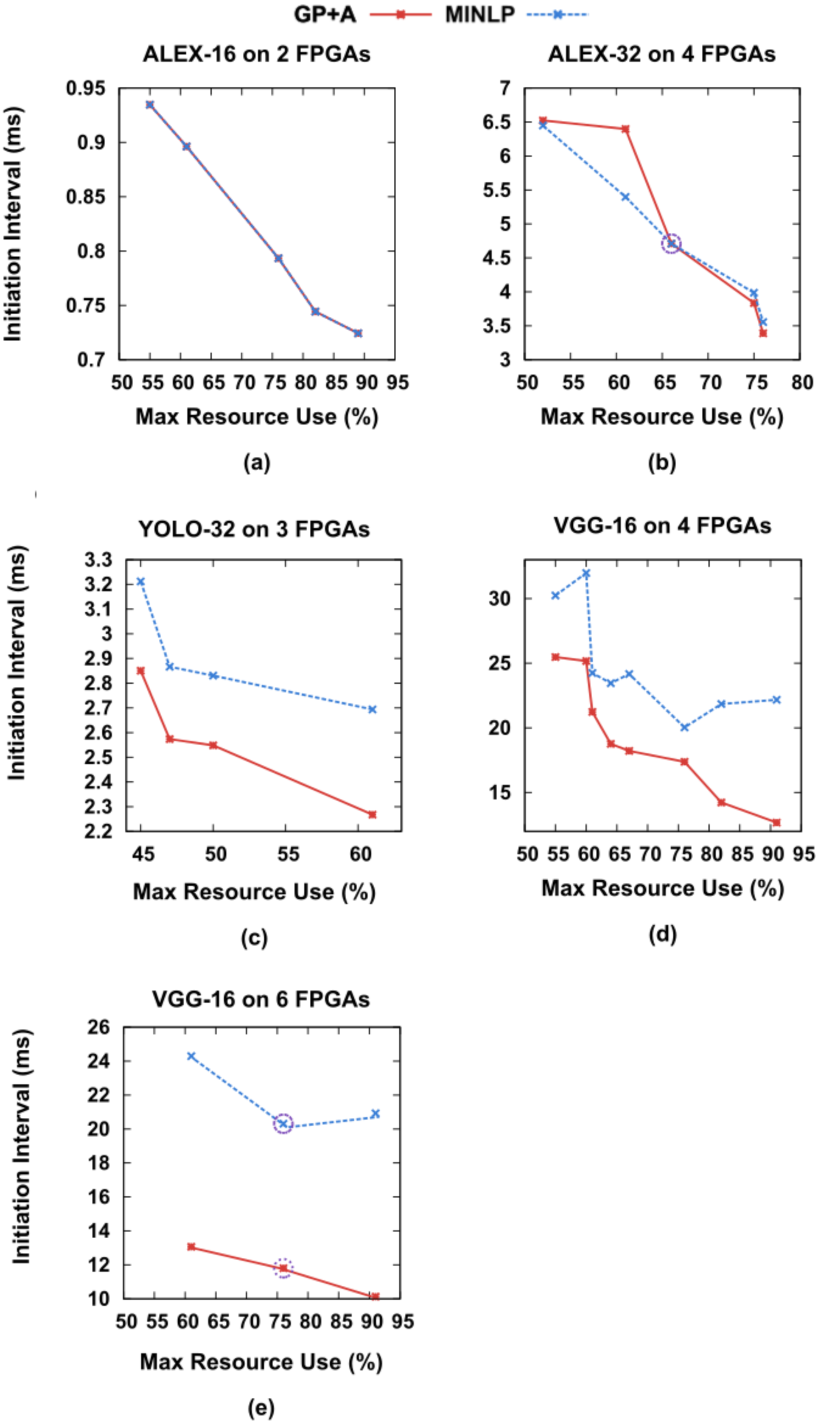


Figure 4.10: Initiation interval as a function of FPGA resource usage: (a) ALEX-16 on 2 FPGAs, (b) ALEX-32 on 4 FPGAs, (c) YOLO-32 on 3 FPGAs, (d) VGG-16 on 4 FPGAs and (e) VGG-16 on 6 FPGAs.

The histograms in Figure 4.9 and Figure 4.11 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 Figure 4.10(b) and Figure 4.10(e), respectively.

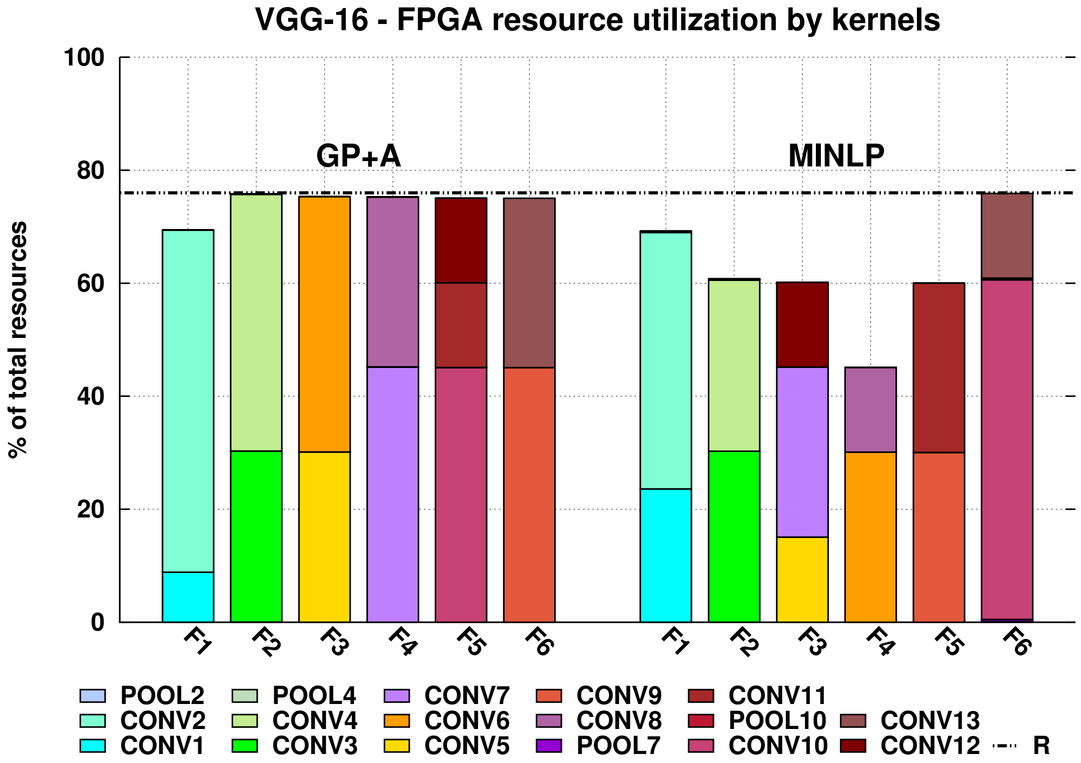


Figure 4.11: VGG-16 kernel allocation on 6 FPGA using GP+A and MINLP.

Figure 4.9 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 Figure 4.11 the allocations are significantly different. While GP+A manages 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.

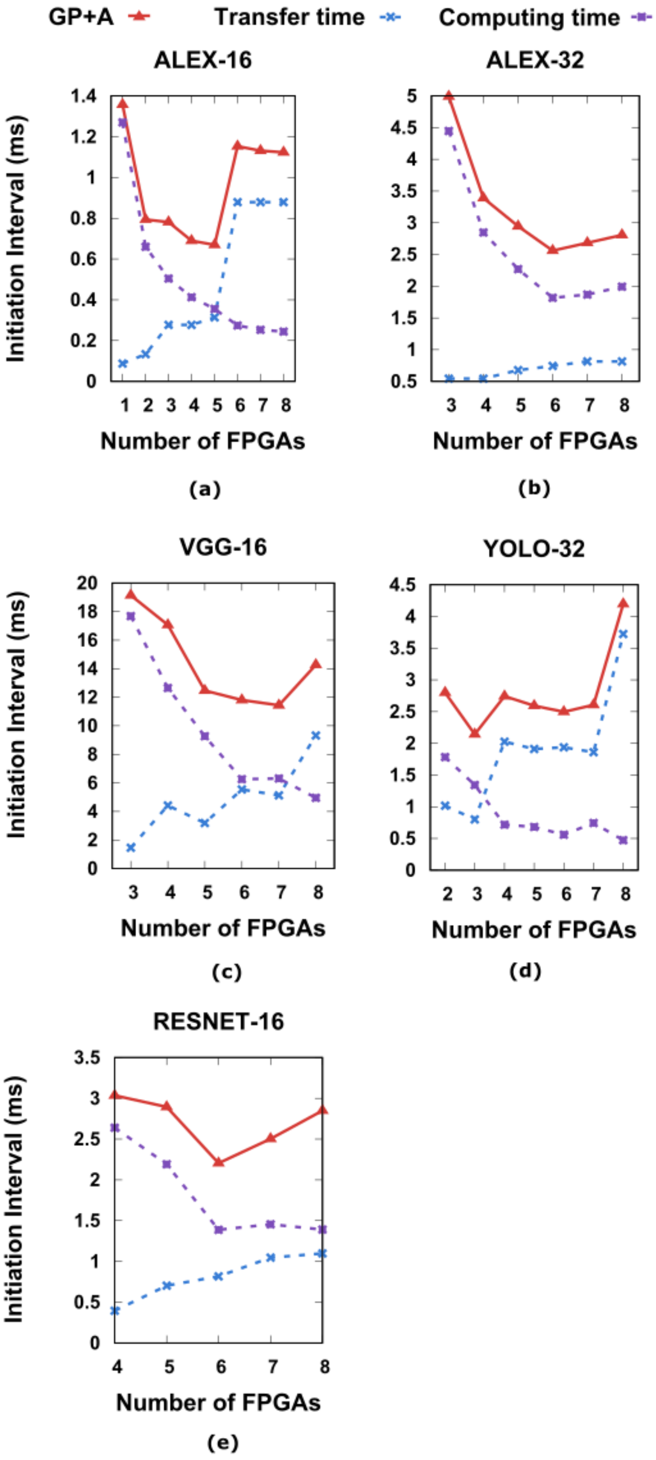


Figure 4.12: Initiation interval as a function of the number of FPGAs used: (a) ALEX-16, (b) ALEX-32, (c) YOLO-32, (d) VGG-16, and (e) RESNET-16.

In Figure 4.12, 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.

4.5 Result Comparison

In this section, we will compare the result obtained using the simplified model proposed in chapter 3, which is denoted by "Simplified-model" and the result obtained in the current chapter, which is called "Enhanced-model". From the formulation of the models, we know that the Enhanced-model is more precise and it includes the optimization of the data transfer time between host CPU and FPGAs. So the data transfer is minimized by allocated as many as possible the consecutive kernels that have a large amount of data flow on a single FPGA. While the Simplified-model does not consider that.

Figure 4.13 shows the allocation of kernels of VGG-16 on 6 FPGAs with a resource usage at 76%. We can clearly see that The Enhanced-model grouped the consecutive kernels together, then allocate them on the same FPGA. In the Simplified-model, the kernels are scattered on 6 FPGAs. It's not difficult to guess the II obtained from Enhanced-model is much better than the Simplified-model with II equals to 12ms and 28.8ms, respectively.

Since the two models are different, the CPU time spent for the two heuristic methods reported in chapter 3 and chapter 4 is also different. For the Alex-16, the CPU time is 0.78s and 25s for the Simplified-model and the Enhanced-model, respectively. In the case of VGG-16, it takes 4.4s for Simplified-model and 66s for the Enhanced-model.

The difference is caused by the data transfer time. In summary, the Simplified-model can be used for multi-kernel applications that do not have a large amount of intermediate data that need to be transferred, i.e. where the data transfer time is much less than the computation time. Instead, the Enhanced-model can be used for any multi-kernel application where the workload can be arbitrarily parallelized.

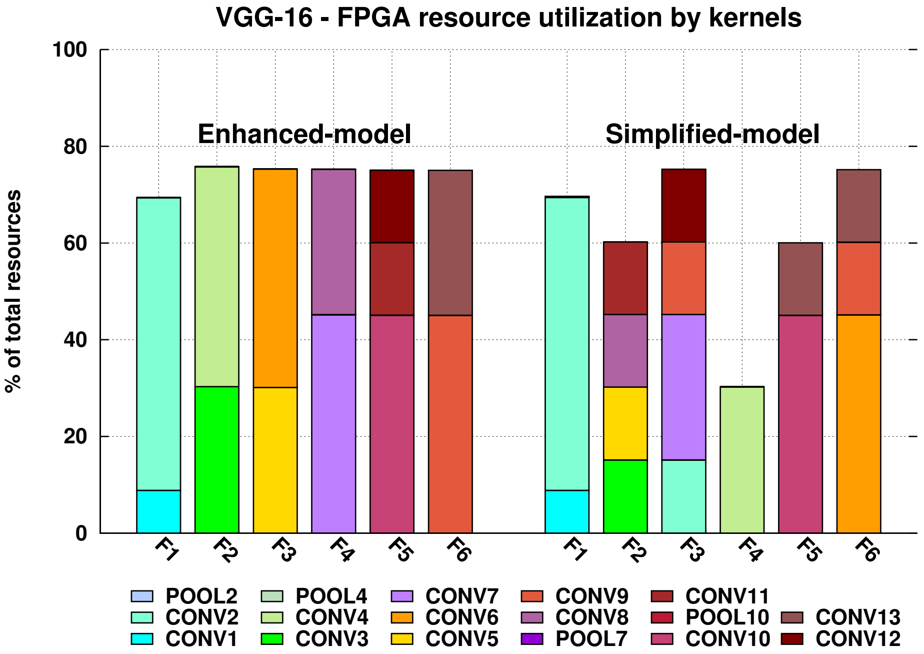


Figure 4.13: Allocation comparison between the simplified and enhanced performance optimization model.

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

Chapter 5

Power-Optimal Mapping of Multi-kernel Applications to Multi-FPGA Platforms

Multi-FPGA platforms like Amazon Web Services F1 are perfect to accelerate multi-kernel pipelined applications, like Convolutional Neural Networks (CNNs). To reduce energy consumption, we propose to upload at runtime the best power-optimized CNN implementation for a given throughput constraint. Our design method gives the best number of parallel instances of each kernel, their allocation to the FPGAs, the number of powered-on FPGAs and their clock frequency. This is obtained by solving a mixed-integer, non-linear optimization problem that models power and performance of each component, as well as the duration of the computation phases—data transfer between a host CPU and the FPGA memory (typically DDR), data transfer between DDR and FPGA, and FPGA computation. The results show that the power saved compared to simply clock gating the fastest implementation is obviously very high, but it is also much more significant than simply scaling the frequency of the fastest implementation or replicating the slowest implementation on multiple FPGAs.

The work presented in this chapter was published in [60].

5.1 Introduction

In this chapter we provide a model to optimize the power consumption of DNNs or other kind of multi-kernel applications on data center platforms including FPGAs, like the AWS F1 instances. Data center workloads vary and accelerators designed for the highest application throughput may be underutilized most of the time, wasting both FPGA resources and energy. Clock gating and frequency scaling can lower energy consumption, but FPGA reconfiguration adapted to application throughput can lower it even more. Throughput is the inverse of the Initiation Interval (II), hence a smaller II means a faster throughput.

Figure 5.1 outlines a multi-FPGA platform. Here the host CPU controls eight FPGAs over a PCI-express (PCIe) bus and can quickly (≈100 ms) reconfigure them with one of several configurations generated offline, adapting them to the actual application performance needs. Reconfiguration is most likely infrequent, e.g., once per minute (or hour), but it optimizes the number of active FPGAs and their clock to spare energy.

Energy-per-computation is the product of power times the initiation interval. Hence at fixed II, minimum power is also minimum energy-per-computation. Since we provide the full power and energy-per-computation versus II curves, other choices can be made (e.g., find the best II to minimize energy-per-computation), according to the application requirements.

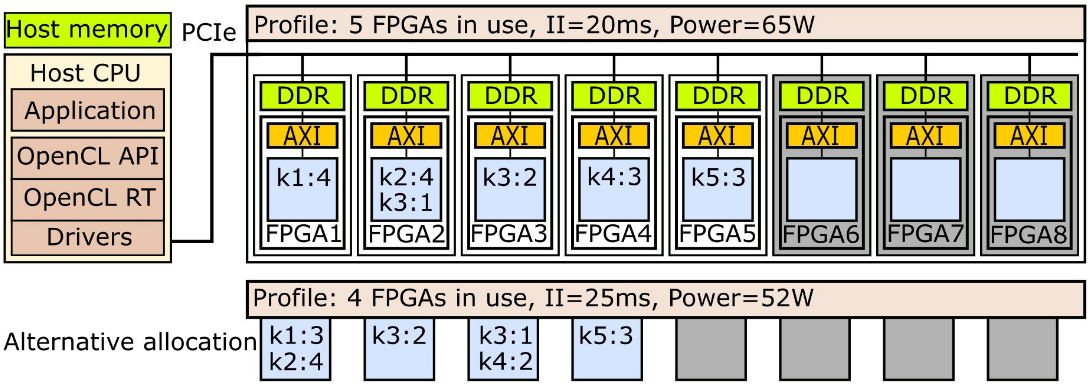


Figure 5.1: Multi-FPGA configurations for different power-performance profiles.

We propose a flow to obtain power-optimized multi-FPGA configuration bitstreams that satisfy different application II requirements. We consider applications that can be modeled as multi-kernel task-level pipelines, and among these we focus our experiments on Convolutional Neural Networks (CNNs). Each task, which corresponds to a CNN layer, can be computed by parallel kernel instances, termed Compute Units (CUs). They are shown in Figure 5.1 as k3:2, k4:3, etc., indicating how many CUs of each kernel are allocated on each FPGA (e.g., k3:2 in FPGA3 means the allocation of two CUs of kernel 3 on it).

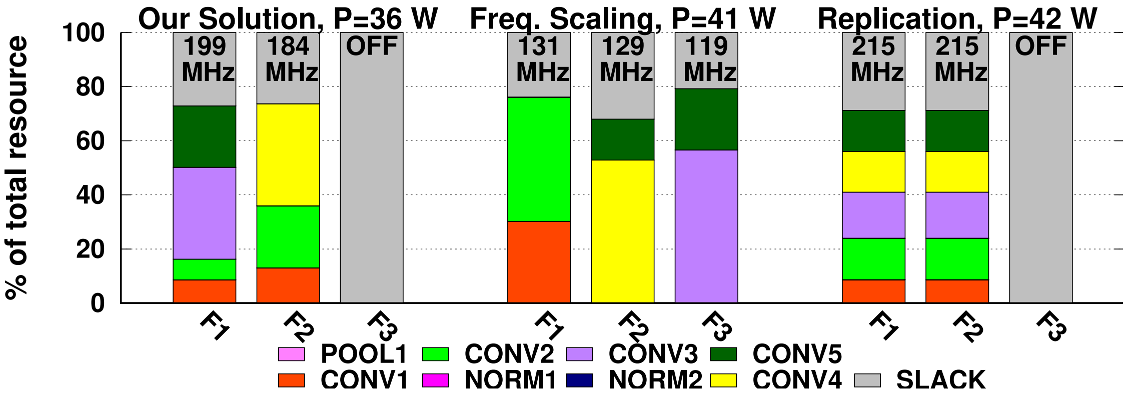


Figure 5.2: Comparison of different power optimization strategies for AlexNet.

After characterizing the multi-FPGA environment and kernels for power, performance, resources, etc., we build a power-performance model that considers both computation and data transfers. Then we solve a Mixed-Integer Non-Linear Problem (MINLP) that, given an II constraint, finds the allocation of the CUs to FPGAs and the clock frequency of each FPGA that minimize power and so also energy per computation.

We compare this strategy to two alternatives: 1) finding the fastest multi-FPGA implementation and applying frequency scaling to reduce energy when the II requirement decreases; 2) finding the fastest single-FPGA implementation and replicating it on the minimum number of FPGAs needed to meet the II constraint. Figure 5.2 compares the three strategies showing an allocation example for AlexNet [5] convolutional layers. The fastest solution (not shown in figure) achieves $II = 0.8\,\text{ms}$ with three FPGAs (F1–F3) each running at a fast and individually optimized clock frequency. But if the application requires $II = 1.4\,\text{ms}$, frequency scaling applied to the fastest solution (middle) consumes 14% more power than an optimized configuration (left), which uses only two FPGAs (F1, F2) at a higher clock frequency. The replication solution (right) is also less efficient and consumes 17% more power than our solution.

5.2 Multi-FPGA Power Optimization

We model CNN layers as K kernels organized in a linear pipeline, including Data Transfer (DT) stages between the host CPU and the FPGA DDRs (see Figure 5.3). The slowest stage sets the II of the pipeline (here the bottleneck is k1, but it could also be DT). To reduce II, we split the kernel workloads into one or more CUs running concurrently, like OpenCL workgroups or CUDA thread groups (see Figure 5.4(a)), and allocate them to the FPGAs (see Figure 5.4(b)). This execution model is well supported by commercial FPGA design tools, e.g., Xilinx SDAccel [12], and it approximately divides the computation time by N when allocating N CUs

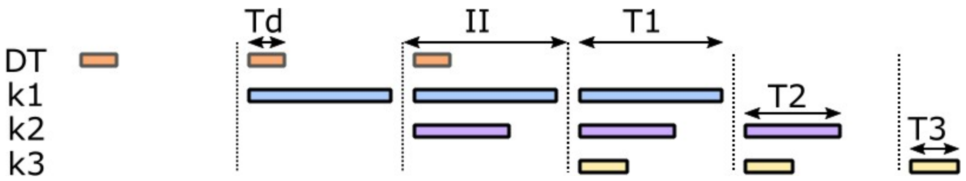
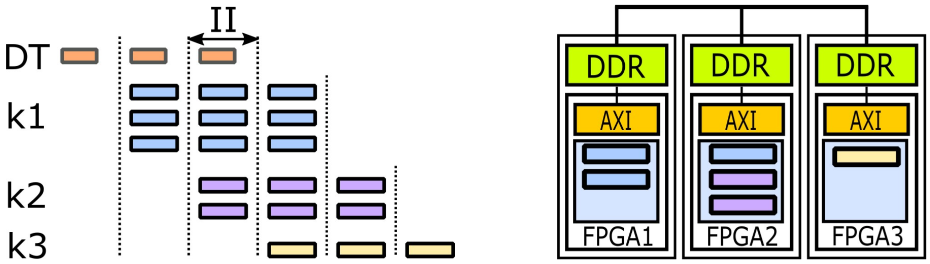


Figure 5.3: CNNs modeled as pipelines of kernels, including data transfer DT.



(a) (b)

Figure 5.4: Kernels are split into multiple compute units allocated on FPGAs.

to each layer (data transfer times are accounted for separately in our model). We design a custom IP for each layer grouping convolution, pooling, and normalization in a single kernel.

Power consumption depends on the number of CUs of each kernel and their allocation to FPGAs. We seek the solution that minimizes power for a given II. Since the II target can change at runtime, we find the optimal solution for each II value in a discretized range. Figure 5.5 shows the proposed design flow. From a C++ or OpenCL high-level description of kernels, we use Xilinx SDAccel to profile their implementation: FPGA resource utilization (LUTs, FFs, DSPs, BRAMs), DDR memory bandwidth, execution time, etc. We enter the profile and target platform characteristics (AWS F1 x8.large in our experiments, which is the largest publicly available cloud FPGA platform) into our power and performance model, then use a MINLP solver to find the configuration with minimum power for each value of II (the points in the graph inset in Figure 5.5). Finally, for each configuration we generate the configuration bitstreams.

5.2.1 Problem Formulation

We aim to minimize the total power while keeping the initiation interval II shorter than II_{max} to satisfy the required throughput (5.1). As shown later, II depends on the number $n_{k,f}$ of CUs of each kernel k allocated to each FPGA f, and on the clock frequency Fck_f of each FPGA. Each CU of kernel k requires $R_{k,t}$ resources of type t (where $t \in$\{FF, LUT, DSP, BRAM, DDR bandwidth\}) and must not exceed the available amount on each FPGA R_t (5.2), while the clock Fck_f of any FPGA f must be slower than the maximum supported FCK (5.3). Moreover,

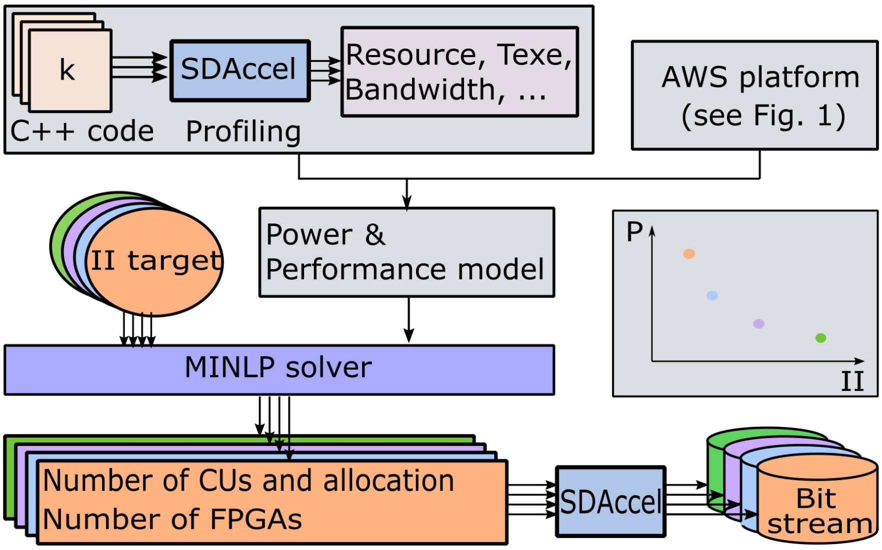

k
C++ code
SDAccel
Profiling
Resource, Texe, Bandwidth, ...
AWS platform (see Fig. 1)
II target
Power & Performance model
P
II
MINLP solver
Number of CUs and allocation
Number of FPGAs
SDAccel
Bit stream

Figure 5.5: Design flow to obtain the power-optimal FPGA configurations.

each kernel k must run on at least one CU (5.4).

$$\begin{aligned} II &\leq II_{max} && && (5.1)\\ \sum_k n_{k,f} R_{k,t} &\leq R_t, && \forall f,\ \forall t && (5.2)\\ Fck_f &\leq FCK, && \forall f && (5.3)\\ N_k = \sum_{f=1}^{F} n_{k,f} &\geq 1, && \forall k. && (5.4) \end{aligned}$$

The resulting problem is a MINLP one because it includes integer ($n_{k,f}$) and real (Fck_f) variables and non-linear constraints.

5.2.2 Initiation Interval (II) Modeling

The top-level computation consists of pipelined data transfers and kernel executions. We use double buffers in the FPGA DDR so that execution can overlap data transfer with the host CPU (using single-buffering requires just a simple change of our model, and it will not be discussed further).

II is limited by the maximum among the data transfer time from host CPU to FPGA DDR T_{h2f} and back T_{f2h}, and the CU execution time T_{exe}. Execution can overlap with data transfer (Figure 5.3), but all data transfers are managed by the CPU, hence

$$II \geq \max(T_{h2f} + T_{f2h}, T_{exe}). \tag{5.5}$$

We now analyze separately the terms in (5.5).

Execution Phase

We assume that kernel workloads are arbitrarily parallelizable via *doall* top-level loops, which is applicable not only to CNNs but also to many machine learning, big data, and scientific applications, and is well supported by the OpenCL and CUDA models of computation. If $T_{wc,k}$ is the *single-CU* execution time of kernel k at maximum FPGA frequency FCK, and the kernel workload is split over $n_{k,f}$ CUs on one or several FPGAs f, then the *actual* kernel execution time in FPGA f, $T_{k,f}$, scales with the number N_k of CUs and the actual frequency Fck_f of FPGA f (5.7), and T_{exe} is the maximum across all kernels and FPGAs (5.8)

$$\begin{aligned}
\delta_{k,f} &= \begin{cases} 1 & \text{if } n_{k,f} > 0 \\ 0 & \text{otherwise} \end{cases}, \qquad \forall f,\ \forall k & (5.6)\\
T_{k,f} &= \tfrac{T_{wc,k}}{N_k} \cdot \tfrac{FCK}{Fck_f} \cdot \delta_{k,f}, \qquad \forall f,\ \forall k & (5.7)\\
T_{exe} &= \max_{k,f} T_{k,f}. & (5.8)
\end{aligned}$$

Here, the allocation variable $\delta_{k,f}$ is 1 if at least one CU of kernel k is allocated to FPGA f, and 0 otherwise.

Host-to-FPGA and FPGA-to-Host Phases

T_{h2f} is the ratio between the total size of input data from the host memory to the DDR of the FPGAs, DI_{H2F}, and the PCIe bandwidth, $\mathbf{B}_{H2F}$. Similarly, T_{f2h} is the ratio between the total size of output data from the DDR of the FPGAs to the host memory, DO_{F2H}, and the PCIe bandwidth, $\mathbf{B}_{F2H}$

$$T_{h2f} = \tfrac{DI_{H2F}}{\mathbf{B}_{H2F}}, \quad T_{f2h} = \tfrac{DO_{F2H}}{\mathbf{B}_{F2H}}. \tag{5.9}$$

In this paper we assume the worst case, namely that direct data transfers between FPGA DDRs are not supported, since this is the case for the AWS F1 platform (again, relaxing this assumption requires a minor change to the model, and will not be discussed further). We also assume that all CUs need the entire input data set $\mathbf{DI}_k$, which is true for CNNs and can be a worst-case assumption for other applications. Hence, we must replicate the input data if the CUs of a kernel k are allocated to multiple FPGAs, and the replication factor α_k is

$$\alpha_k = \textstyle\sum_{f=1}^{F} \delta_{k,f}, \qquad \forall k. \tag{5.10}$$

The data transferred in the host-to-FPGA phase amount to

$$DI_{H2F} = \textstyle\sum_{k=1}^{K} \alpha_k \mathbf{DI}_k. \tag{5.11}$$

Note that constant data (e.g., weights and bias in CNNs) are not considered, since they are transferred once during initialization.

Differently from the input data, we assume instead that the output data computed by a kernel, $\mathbf{DO}_k$, are equally divided among its CUs, hence we transfer

$$DO_{F2H} = \sum_{k=1}^{K} \mathbf{DO}_k. \tag{5.12}$$

5.2.3 Power Modeling

FPGA-related average power consumption has a constant *static* contribution, P_s, and a *dynamic* one, P_d, accounting for both data transfer with the host and the FPGA processing. Table 5.1 shows all the variables involved in the power model.

DDR Power Model

Table 5.1: Variables used in power model equations.

Notation	**Description**
P_{total}	total power
Fck_f	actual working frequency of FPGA f
T_{h2f}, T_{f2h}	data transfer time host to DDR and DDR to host, resp.
T_{exe}	execution time
E_{h2f}, E_{f2h}	energy spent during T_{h2f} and T_{f2h}, resp.
E_{exe}	total energy spent during T_{exe}
E_{rw}	energy spent by accesssing to DDR during T_{exe}
DI_{H2F}	total data transferred in the host-to-FPGA phase
DO_{F2H}	total data transferred in the FPGA-to-host phase
P_{DDRdr}, P_{DDRdw}	DDR dynamic power when reading and writing, resp.
P_{fs}	on-chip static power
$P_{fd,f}$	dynamic power of FPGA f

we obtained the FPGA DDR power using a calculator [61] and from experiments on the AWS F1 platform, which includes an API to report power consumption. Idle DDR consumes only static power, P_{DDRs}, while dynamic power depends on the normalized read B_r and write B_w bandwidths (i.e., $B_r = 1$ if the maximum bandwidth is used for reading), and is the sum of the corresponding P_{DDRdr} and P_{DDRdw}. The equations that we used, with coefficients expressed in Watt and coming from the characterization above, are: $P_{DDRs} = 0.5$, $P_{DDRdr}(B_r) = 0.672B_r$ and $P_{DDRdw}(B_w) = 0.4B_w$.

Single FPGA Power

it consists of static and dynamic power obtained with the Xilinx Power Estimator (XPE) [62]. Static power P_{fs} includes logic P_{fls} and memory I/O P_{fio} power

$$P_{fs} = P_{fls} + N_{fio} \cdot P_{fio} \tag{5.13}$$

with $N_{fio} = 4$ DDR banks per FPGA. One I/O bank consumes $P_{fio} = 0.414\,\mathrm{W}$ from [62] and logic power is $P_{fls} = 2.842\,\mathrm{W}$.

Dynamic power $P_{fd,f}$ of FPGA f depends on each kernel's CUs allocated to f, $n_{k,f}$, and scales with clock frequency

$$P_{fd,f} = \textstyle\sum_k n_{k,f} \cdot P_k \cdot \frac{Fck_f}{FCK} \tag{5.14}$$

with P_k the dynamic power of one CU of kernel k when running at the highest clock frequency FCK.

Multi-FPGA Power

static power P_s depends only on the number of active FPGAs, N_F:

$$P_s = N_F \left(P_{DDRs} + P_{fs}\right). \tag{5.15}$$

Total dynamic power P_d depends on the energy spent during data transfer from host to FPGA DDRs E_{h2f}, processing E_{exe}, and data transfer from FPGA DDRs to host E_{f2h}

$$P_d = \frac{E_{h2f}+E_{exe}+E_{f2h}}{II} = \frac{E_d}{II}. \tag{5.16}$$

Here, E_{h2f} depends on the data replication factor α_k (5.10), DDR write bandwidth Bw_k, and transfer time tw_k obtained from the FPGA profiling reports

$$E_{h2f} = \textstyle\sum_k^K \alpha_k \cdot P_{DDRdw}(Bw_k) \cdot tw_k. \tag{5.17}$$

Similarly, E_{f2h} depends on DDR read bandwidth Br_k, and transfer time tr_k also from profiling (note that there is no output data replication)

$$E_{f2h} = \textstyle\sum_k^K P_{DDRdr}(Br_k) \cdot tr_k. \tag{5.18}$$

CU execution energy E_{exe} includes the energy to read/write data on DDR E_{rw} and the FPGA processing energy E_c

$$\begin{aligned} E_{rw} &= \textstyle\sum_k^K N_k \left(P_{DDRdr}(br_k) + P_{DDRdw}(bw_k)\right) \cdot T_{exe} && (5.19)\\ E_c &= \textstyle\sum_f^F P_{fd,f} \cdot T_{exe} && (5.20)\\ E_{exe} &= E_{rw} + E_c. && (5.21) \end{aligned}$$

where the data transfer bandwidths that the CUs of kernel k use to read from and write to DDR are br_k and bw_k, respectively.

Total power consumption P_{total} is given by the static power from (5.15) and the dynamic power from (5.16)

$$P_{total} = P_s + P_d. \tag{5.22}$$

The energy per computation is $E_{comp} = P_{total} \cdot II$.

5.3 Experiments

We check our optimization method against *frequency scaling*, *clock gating*, and *replication* for two widely used and realistically large CNNs, AlexNet [5] and VGGNet [6]. However, note that our technique is not specific to CNNs (even though we evaluate it for well-known CNNs), and only depends on the assumption of *arbitrarily parallelizable kernel pipelines mapped to multiple FPGAs with DDR-based memory transfers*. We show results for AlexNet using 32-bit floating-point and 16-bit fixed-point, and VGGNet using only fixed-point. To solve the minimization problem in (5.1)–(5.3) we use a state-of-the-art MINLP solver, Couenne [2]. Note that a MINLP problem can have in principle multiple local minima. We tried running the solver multiple times, but the result was always the same.

We characterize the kernels for the power and performance model discussed in Sec. 5.2 using the FPGA profiling reports from Xilinx SDAccel and actual measurements on an Amazon AWS F1 x8-large instance with eight Xilinx UltraScale+ FPGAs, each with four DDR banks and a PCIe connection to the host CPU (see Figure 5.1). MINLP and SDAccel run on CentOS Linux 6.9 on a 16-core Intel Core i7-6900K @3.2GHz. We are using one processor for our experiments.

The MINLP solver requires ≈1 hour to optimize one AlexNet fixed-point implementation, ≈1 day for AlexNet floating-point, and ≈30 hours for VGGNet. Faster heuristics are left to future work.

Characterization data from AWS executions for the AlexNet and VGG benchmarks are shown in Table 5.3-Table 5.4, respectively. The performance of the optimization methods is shown Figure 5.7, and the number of FPGAs used as a function of the II is shown in Figure 5.6. The labels in the figure captions are:

- *Our Solution* is the MINLP optimum using the design flow in Figure 5.5; note that this is most likely a local minimum, since the optimization space is not convex;

- *Freq. Scaling* scales down the clock frequency of the fastest-II MINLP solution to meet each actual II requirement; note that the AWS platform does not support voltage scaling; including it into our model would be a simple modification;

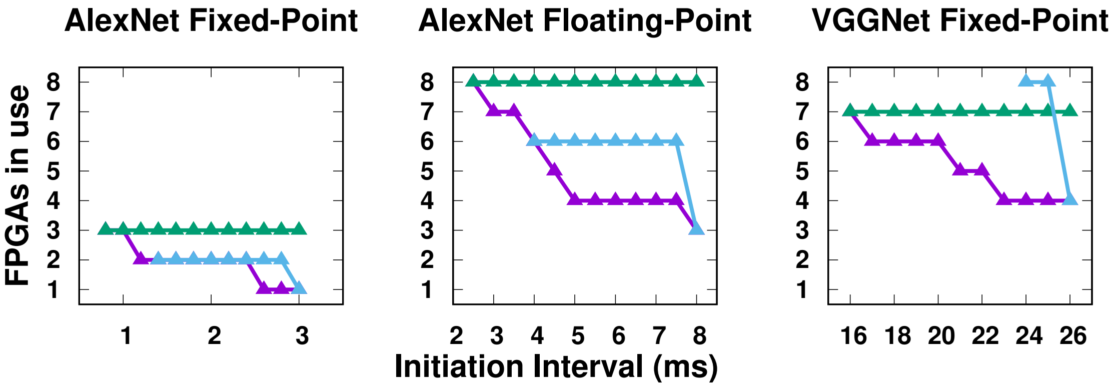


Figure 5.6: Number of used FPGAs as a function of the optimization method (FS is frequency scaling, CK is clock gating) and the target initiation interval.

Table 5.2: AlexNet 32-bit floating-point kernel characterization. Layers: convolutional Conv, pooling Pool.

Kernels	BRAM (%)	DSP (%)	T_{wc} (ms)	Bw/Br (%)	tw/tr (ms)	bw/br (%)	P_k (W)
Conv1	13.07	21.24	13	26.54 / 21.96	0.2 / 0.55	0.193 / 0.130	4.542
Pool1	2.84	0	1.78	41.48 / 13.62	0.29 / 0.21	1.415 / 0.341	0.633
Norm1	6.1	2.11	0.839	12.5 / 11.39	0.23 / 0.26	0.725 / 0.725	1.091
Conv2	8.73	37.59	7.19	8.45 / 9.4	0.35 / 0.19	0.54 / 0.052	8.367
Norm2	7.75	2.11	0.807	9.32 / 9.92	0.19 / 0.18	0.466 / 0.466	1.252
Conv3	5.22	28.13	7.78	7.92 / 26.33	0.23 / 0.1	1.18 / 0.072	6.173
Conv4	2.13	37.5	9.08	6.63 / 12.7	0.4 / 0.21	1.063 / 0.073	7.979
Conv5	8.73	37.5	4.84	3.77 / 4.38	0.38 / 0.09	1.027 / 0.017	8.150

- *Clock Gating* stops the FPGA clock when the CUs of the fastest-II MINLP solution finish computation; note that the AWS platform does not support power gating at runtime; since static power is ≈20% of the total power, considering power gating would bring *Clock Gating* closer to *Freq. Scaling*, but still far from *Our Solution*.

- *Replication* makes copies of the MINLP solution that uses the minimal number of FPGAs (hence the slowest solution) until it meets the II requirement.

By design, 1) all plots in Figure 5.7 except for *replication* start at the best performance point, and 2) *replication* meets *our solution* at the worst performance point. Note that *our solution* always yields equal or superior results to other methods.

Freq. scaling and *clock gating* show a reciprocal dependency between power and II, because they keep the same number of kernel CUs, the same FPGA allocation,

Table 5.3: AlexNet 16-bit fixed-point kernel characterization. Layers: convolutional Conv, pooling Pool.

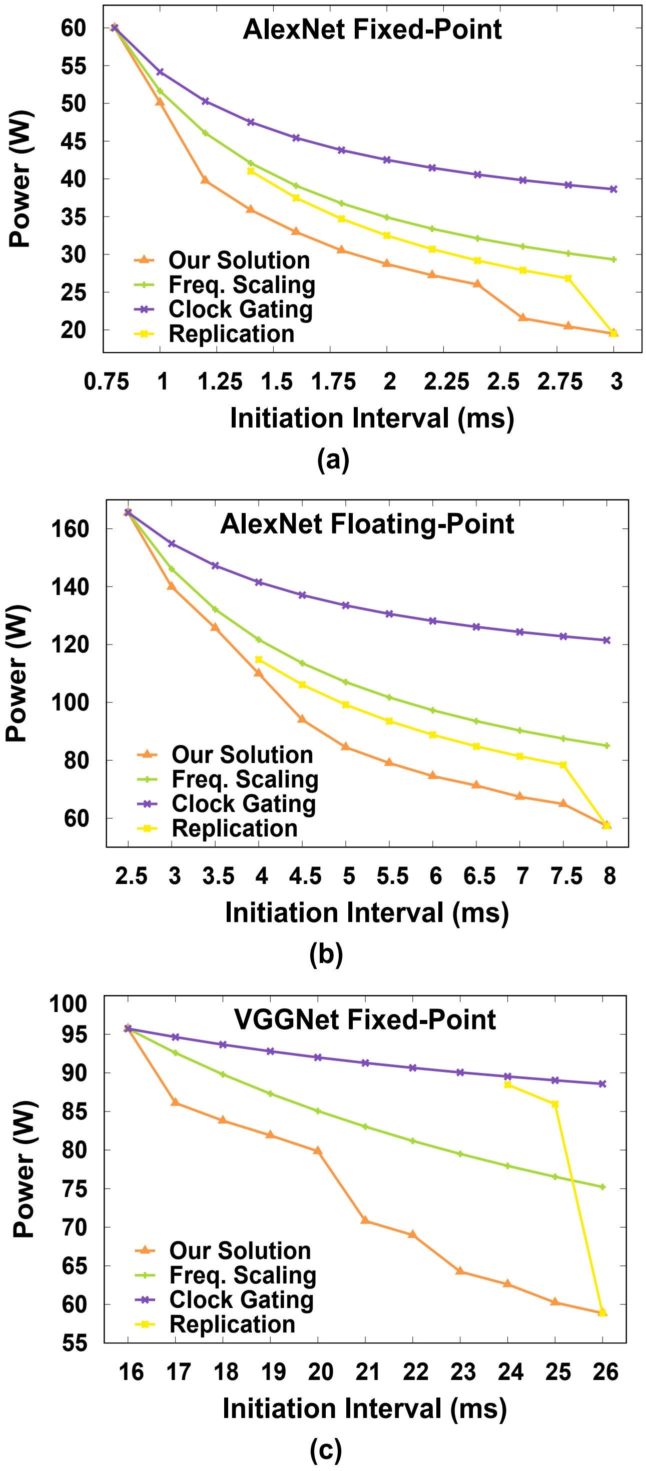
Kernels	**BRAM** (%)	**DSP** (%)	$\mathbf{T_{wc}}$ (ms)	**Bw/Br** (%)	**tw/tr** (ms)	**bw/br** (%)	$\mathbf{P_k}$ (W)
Conv1	10.59	4.31	5.16	16.19 / 15.42	0.2 / 0.39	0.209 / 0.052	1.004
Pool1	0.05	0	1.78	26.86 / 8.79	0.23 / 0.17	0.709 / 0.171	0.605
Norm1	2.53	0.06	0.78	6.26 / 9.62	0.23 / 0.15	0.389 / 0.388	0.596
Conv2	4.39	7.63	4.11	7.08 / 7.77	0.22 / 0.12	0.475 / 0.046	1.438
Norm2	6.66	0.06	0.67	4.59 / 7.63	0.2 / 0.12	0.281 / 0.279	0.664
Conv3	2.63	5.66	6.7	1.864 / 14.4	0.5 / 0.09	0.684 / 0.042	1.109
Conv4	1.91	7.55	5.06	5.346 / 14.5	0.256 / 0.09	0.737 / 0.056	1.274
Conv5	4.39	7.55	3.29	2.93 / 2.08	0.24 / 0.09	0.755 / 0.012	1.340

Table 5.4: VGGNet 16-bit fixed-point kernel characterization results

Kernels	**BRAM** (%)	**DSP** (%)	$\mathbf{T_{wc}}$ (ms)	**Bw/Br** (%)	**tw/tr** (ms)	**bw/br** (%)	$\mathbf{P_k}$ (W)
Conv1	3.67	2.95	28.8	17.33 / 24.79	0.18 / 2.70	0.028 / 0.484	0.914
Conv2	9.97	15.14	67.8	83.20 / 22.78	0.80 / 2.94	0.321 / 0.206	2.106
Pool2	11.6	0.03	13.3	83.86 / 23.33	0.80 / 0.72	1.045 / 0.261	0.825
Conv3	9.97	15.14	22.7	49.28 / 24.47	0.34 / 1.37	0.269 / 0.307	2.108
Conv4	9.97	15.14	32.1	78.28 / 24.26	0.46 / 1.38	0.380 / 0.217	2.107
Pool4	2.94	0.03	6.9	72.86 / 22.67	0.92 / 0.74	1.020 / 0.254	0.714
Conv5	8.32	15.07	22.8	23.37 / 22.52	0.36 / 0.74	0.341 / 0.153	2.055
Conv6-7	8.32	15.05	32.9	44.44 / 23.33	0.38 / 0.72	0.472 / 0.106	2.063
Pool7	1.50	0.03	3.5	56.74 / 17.18	0.29 / 0.24	0.985 / 0.246	0.615
Conv8	2.12	15.02	24.5	8.986 / 19.06	0.47 / 0.44	0.455 / 0.071	1.982
Conv9-10	2.12	15.02	37.7	10.93 / 19.28	0.77 / 0.43	0.590 / 0.046	1.979
Pool10	0.05	0.01	2.1	31.87 / 11.84	0.26 / 0.18	0.800 / 0.200	0.582
Conv11-13	2.12	14.99	20.3	3.319 / 10.96	0.63 / 0.19	0.629 / 0.022	1.986

and the same number of active FPGAs. They satisfy the II constraint either by scaling the FPGA clock frequency only, or by disabling the clock in addition to scaling it. Unlike them, *our solution* saves more power when the II constraint is relaxed, because it optimizes both the number and allocation of CUs, the number of active FPGAs, and their working frequencies *at the same time*.

Replication starts from the optimal results obtained using *our solution* for the highest II. For both AlexNet implementations in Figure 5.7(a) and Figure 5.7(b),



(a)

(b)

(c)

Figure 5.7: Power versus initiation interval in (a) AlexNet Fixed-Point, (b) AlexNet Floating-Point, and (c) VGGNet Fixed-Point.

replication finds better solutions than *freq. scaling* and *clock gating* for II constraints when the execution time is much higher than data transfer time. In fact, from (5.7) the execution time is inversely proportional to CU number, while from (5.9)–(5.11) T_{h2f} is proportional to the number of CUs, because the input data are replicated with the same factor. VGGNet [Figure 5.7(c)] shows an extreme case when data transfer time is very high and solution replication mostly increases power by increasing the number of CUs, without a significant reduction of the II, since it is dominated by data transfer time. However, as shown in Figure 5.2, *our solution* smartly groups the kernel CUs on fewer FPGAs, minimizing both data transfer time and power at the same time.

5.4 Conclusion

We proposed a power-performance optimization method to optimally configure a multi-FPGA platform running multi-kernel pipelined workloads. Given an II target, the solution of a MINLP problem provides an optimal allocation of the best number of CUs for each kernel so as to minimize the overall power consumption. Compared to applying frequency scaling to reduce both II and power starting from a fast configuration, or to replicating a slow configuration on multiple FPGAs, our solution provides a much more effective way of saving power.

Chapter 6

Fast Power-Optimal Multi-Kernel Application Allocation on Multi-FPGA Platforms

In Chapter 5 we proposed a power optimization model to find the power minimum solution for given throughput constraint offline and upload the specific bitstream at runtime to program the FPGA. The model has been solved using a MINLP solver; however, it is not feasible to use it for design space exploration due to its inefficiency.

In this chapter, we provide two heuristic optimization methods that improve result quality within a bounded time. We use several very large designs to demonstrate that both heuristics obtain comparable results to MINLP, when it can find the best solution, and they obtain much better results than MINLP, when it cannot find the optimum within a bounded amount of time. The heuristic methods can also be thousands of times faster than the MINLP solver.

This work has been submitted to *IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems.*

6.1 Power Modeling

The average FPGA power consumption has a constant *static* component, P_s, and a *dynamic* one, P_d, including both the data transfer with the host and the FPGA processing. Total power consumption is thus

$$P_{total} = P_s + P_d. \tag{6.1}$$

The detailed power model is discussed in Chapter 5.

Static Power

includes the DDR static power, P_{DDRs}, and the FPGA static power, P_{fs}. In addition, it is proportional to the number of active FPGAs, F

$$P_s = F\left(P_{DDRs} + P_{fs}\right). \tag{6.2}$$

Dynamic Power

is proportional to the average dynamic energy, E_d, spent during one II

$$P_d = \tfrac{E_d}{II}. \tag{6.3}$$

Dynamic energy consists of DDR dynamic energy, E_{ddrd}, due to data transfer between host and DDR, and the processing energy, E_c, due to the CUs allocated on FPGA f

$$\begin{aligned} E_d &= E_{ddrd} + E_c && (6.4)\\ E_c &= \textstyle\sum_f^F P_{fd,f} \cdot T_{exe} && (6.5)\\ P_{fd,f} &= \textstyle\sum_k n_{k,f} \cdot P_k \cdot \frac{Fck_f}{FCK}. && (6.6) \end{aligned}$$

The dynamic power of FPGA f, $P_{fd,f}$, depends on the number of CUs of each kernel allocated to it, $n_{k,f}$, and scales with the clock frequency. The detailed equation for the calculation of the DDR dynamic power is discussed in Chapter 5.

6.2 Heuristic Solutions

The optimization problem in chapter 5 can be solved using a Mixed-Integer Non-Linear Programming (MINLP) solver. However, this may need a very long time to solve, being often impractical, especially for explorations that invoke multiple times the MINLP solver. Hence, we propose two heuristic methods to improve the exploration efficiency. The first still uses a MINLP solver, but over a much smaller exploration space. The second avoids completely the MINLP solver and is much faster.

6.2.1 First Heuristic Solver, Using MINLP

To speed up the MINLP solver, we fix the number of active FPGAs, limit the number of possible CUs for each kernel, and simplify the power model. In our previous work Chapter 5, we empirically noticed that the best solutions save static power by always using the minimum number of FPGAs, F_{min}, most likely because of the static power consumption. Hence, our first heuristics uses F_{min} as a hard bound on resources instead of exploring allocations on more FPGAs. To obtain F_{min}, we first determine the minimum number of CUs for each kernel that satisfies the II_{max} constraint

$$CU_{min_k} = \lceil T_{wc,k}/II_{max} \rceil \tag{6.7}$$

and then we find F_{min} by using the resource utilization

$$F_{min} = \max_t \left\lceil \frac{\sum_{k=1}^{K} R_{k,t} \cdot CU_{min_k}}{R_t} \right\rceil . \tag{6.8}$$

In our experiments with CNNs and NLP, the maximum in (6.8) is always determined by the total number of required and available DSP units (i.e., $R_{k,t}$ and R_t, with $t = \mathrm{DSP}$) on a single FPGA.

We introduce an additional binary variable in the problem, $extraCU_k = \{0, 1\}$, to limit the number of CUs per kernel, CU_k, to at most one higher than CU_{min_k} from (6.7)

$$CU_k = CU_{min_k} + extraCU_k . \tag{6.9}$$

We do this because additional CUs may reduce the execution time of a kernel, which may become closer in speed to other kernels, hence allowing us to reduce the frequency when these kernels are all allocated to the same FPGA. With CU_k limited to only two values in (6.9), the possible values of the allocation variables $n_{k,f}$—the sum of which over the active FPGAs is CU_k—is also largely reduced, which has a substantial effect on the execution time of the MINLP solver.

We do not include the data-transfer power in the model because it is typically much lower than the computation power. In this way, the model is further simplified. Notice, however, that this power contribution will be implicitly minimized by our method, since the CUs of the same kernel are likely allocated in the same FPGA because they have the same execution time. As a result, the input data of that kernel will not be duplicated, and data transfer and its associated power consumption will also be reduced.

6.2.2 Second Heuristic Solver, Without MINLP

For larger problems, even the simplified model introduced in Sec. 6.2.1 can be too slow. To further speed up the solution, we propose another heuristic method that does not rely on external solvers.

For the same reasons explained in Sec. 6.2.1, we determine the number of active FPGAs $F = F_{min}$ as in (6.8), and we determine before the allocation with (6.7) the minimum number of CUs for each kernel, and use the auxiliary binary variable as in (6.9). This leads to 2^K possible combinations for the number of CUs, which we test exhaustively as shown in Algorithm 4 (line 3). Notice that although this is exponential, the number of kernels K is usually small and the run time is short, as we report in the experiments. Moreover, the combinations that do not fit in the FPGAs are pruned early by the filter on line 9 to further reduce the run time.

R_{total} in the pseudo-code refers to the DSP resources needed by each kernel, which limit the allocation of computation-intensive applications, such as CNNs and transformer networks, before other resources are exhausted (LUT, FF, BRAM).

From the solution generated in Chapter 5, we can see that to optimize the power consumption, all the kernels that have similar execution time are allocated on the same FPGA. This way can further reduce the FPGA working frequency, hence, reduce the power consumption. We also design the allocation algorithm to do a similar job. Prior to allocation, we set the FPGA resource slack R according to resource constraints (line 10). Then we sort the kernels (line 12) in descending order of the execution time obtained with CU_{min_k} CUs. In this way, we favor the allocation of kernels with similar execution time to the same FPGA, helping to reduce the operating frequency, power, and input data duplication (lines 14 to 29). At the end of this loop, some kernels might have residual CUs that could not be allocated. Therefore, we go through the kernels in the opposite order, allocating the kernels with the smallest execution time to FPGAs in increasing operating frequency order (lines 32 to 47). Note that the kernels are ordered in descending order of execution time on FPGAs, and they are allocated on FPGA in the same order, so the first FPGA will have the highest frequency, and the last one the slowest frequency. We accept only the allocations with total data transfer times below II_{max} (lines 50 to 53), and we select the allocation with minimum power consumption.

6.3 Experiments

We use the same MINLP solver from Chapter 5 to implement the first heuristic method (see Sec. 6.2.1), and we implemented the second one in C++ (see Sec. 6.2.2). We validate the methods using two widely used CNNs, 8-layer AlexNet [59] (32-bit floating-point, ALEX-32, and 16-bit floating-point, ALEX-16) and 17-layer VGG [6] (16-bit floating-point, VGG-16), and a transformer network for NLP [8] (16-bit fixed-point, one decoder, one encoder, four heads in the attention layer, 11 layers total, TRANSFORMER-16). Table 6.1

Algorithm 4: Second heuristic allocation method

```
procedure AllocateCUs($CU_{min}$, $R_{total}$, $II_{max}$)
    // Vector of min required CUs per kernel
    $CU_{min} = (CU_{1_{min}}, CU_{2_{min}}, \dots, CU_{K_{min}})$
    boolean  $extraCU = (CU_{1_e}, CU_{2_e}, \dots, CU_{K_e})$
    for $c = 1$ to $2^K$ do // $2^K$ combinations in total, cause each kernel has two possible CUs
        assign $extraCU$ according to $c$
        $alloc$ = False;   $R_{total} = 0$
        for $k = 1$ to $K$ do
            $CU_k = CU_{k_{min}} + extraCU_k$
            $R_{total}$ += $CU_k \cdot R_k$
        if $R_{total} < F \cdot R$ then
            // Vector of FPGA resource slack initialized to constraint value
            $S = (S_1, S_2, \dots, S_F)$;   $\forall f: \; S_f = R$
            // Allocated CUs initialized to zero
            $\forall k, f: \; n_{k,f} = 0$
            // Sort by descending exec. time
            sortCU($CU$)
            $k = 0$
            for $f = 1$ to $F$ do
                $R_{acc} = 0$
                while $k < K$ do
                    $R_{acc}$ += $CU_k \cdot R_k$
                    if $R_{acc} \leq R$ then
                        $\delta CU_k = CU_k$
                        $S_f = S_f - CU_k \cdot R_k$
                        $CU_k = 0$
                        $n_{k,f} = \delta CU_k$
                    else
                        $\delta CU_k = \lfloor S_f / CU_k \rfloor$
                        $CU_k = CU_k - \delta CU_k$
                        $S_f = S_f - \delta CU_k \cdot R_k$
                        $n_{k,f} = n_{k,f} + \delta CU_k$
                        break;
                    $k = k + 1$
            if $\sum_k CU_k > 0$ then
                $k = K - 1$
                for $f = F$ to $1$ do
                    $R_{acc} = 0$
                    while $k > 0$ and $CU_k > 0$ do
                        $R_{acc}$ += $CU_k \cdot R_k$
                        if $R_{acc} \leq S_f$ then
                            $\delta CU_k = CU_k$
                            $S_f = S_f - CU_k \cdot R_k$
                            $CU_k = 0$
                            $n_{k,f} = \delta CU_k$
                        else
                            $\delta CU_k = \lfloor S_f / CU_k \rfloor$
                            $CU_k = CU_k - \delta CU_k$
                            $S_f = S_f - \delta CU_k \cdot R_k$
                            $n_{k,f} = n_{k,f} + \delta CU_k$
                            break;
                        $k = k - 1$
            if $\sum_k CU_k = 0$ then
                $alloc$ = True
            Calculate $T_{f2h} + T_{h2f}$
            if $alloc$ and $T_{f2h} + T_{h2f} < II_{max}$ then
                Update $Fck_f$
                Calculate $Power$
        $n_{k,f}$ = arg(min($Power$))
```

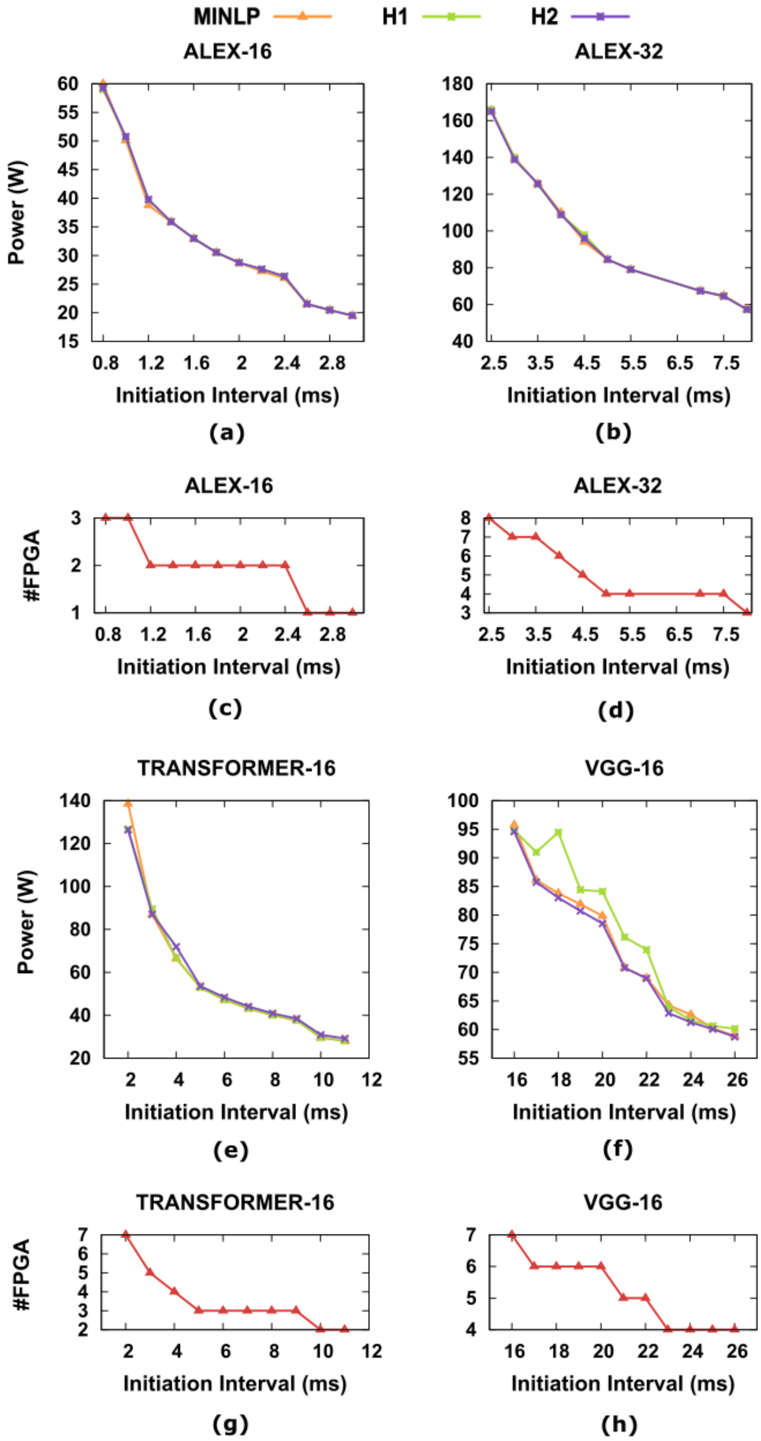


Figure 6.1: Minimum power and number of FPGAs function of the initiation interval obtained with optimization method in Chapter 5 (MINLP), first (H1) and second (H2) heuristic methods for (a) 16-bit floating-point AlexNet, (b) 32-bit floating-point AlexNet, (e) 16-bit fixed-point Transformer, and (f) 16-bit floating-point VGG.

Table 6.1: Transformer kernel characterization on the AWS F1 platform

Kernels	**BRAM** (%)	**DSP** (%)	$\mathbf{T_{wc}}$ (ms)	**Bw / Br** (%)	**tw / tr** (ms)	**bw / br** (%)	$\mathbf{P_k}$ (W)
Attention1	20.9	31.5	9.5	1.23 / 0.39	8.9 / 0.3	1.03 / 0.004	4.12
Attention2	9	16.5	6.3	9.52 / 0.89	0.59 / 0.12	1.03 / 0.004	3.04
feed_forward1	0.9	3.7	16.7	77.4 / 3.63	0.28 / 0.11	0.275 / 0.01	0.85
feed_forward2	0.9	3.7	16.8	11.5 / 0.95	1.94 / 0.11	0.28 / 0.011	0.85
norm	0.5	0.5	0.3	20.5 / 10.24	0.1 / 0.1	0.315 / 0.16	0.92

shows the transformer kernel characterizations on the AWS F1 platform. Kernel characterizations for the other three implementations are reported in Chapter 5.

Figure. 6.1 shows the experimental results obtained on a CentOS Linux v6.9 machine with an Intel Core i7-6900K processor with 128 GB RAM. MINLP denotes the optimization with the method described in Chapter 5, H1 denotes the first heuristic method, and H2 denotes the second heuristic method. Figures. 6.1(a)(b)(e)(f) show the minimum power consumption obtained for a range of II using each optimization method. Figures. 6.1(c)(d)(g)(h) show the corresponding number of FPGAs. For small-size problems, e.g., ALEX-16 and ALEX-32, all optimization methods yield the same result, proving the effectiveness of the proposed heuristics. The latter are much faster than MINLP, which needs around one hour to solve the ALEX-16 compared to 20 min for H1 and a few seconds for H2, or around one day for ALEX-32 compared to less than 15 h for H1, and a few seconds for H2.

For larger problems, e.g., TRANSFORMER-16 in Figure. 6.1(e), MINLP requires one day, while H1 needs 10 h, and H2 always completes in a few seconds. Additionally, the proposed heuristics H1 and H2 find better results, especially for $II = 2\,\text{ms}$. H2 returns a suboptimal solution for $II = 4\,\text{ms}$, but it finishes comparatively much faster, in a few seconds.

For even larger problems, e.g., VGG-16 in Figure. 6.1(f), H1 often misses the best solution by roughly 12 % after running for about 15 h compared to MINLP running for 30 h. H2 solutions are the best with run times around 10 s.

Summarizing, for small problems both our heuristic methods, H1 and H2, find good solutions. H1 may even obtain better solutions, such as for $II = 4\,\text{ms}$ for TRANSFORMER-16. For larger problems, H2 is better and much faster.

Note that for larger problems we had to stop MINLP early. Hence, the results in Figure. 6.1(f) are the best after 24 h to 30 h, but still sub-optimal. This explains why the proposed heuristic methods H1 and H2 may reach better results (e.g., for VGG-16).

The number of FPGAs used for the best solution achieved using the three different methods are shown in Figures. 6.1(c)(d)(g)(h). In all cases, the optimal number of FPGAs is the same.

6.4 Conclusion

We propose two heuristic methods to efficiently obtain energy-optimal or near-optimal solutions to configure a multi-FPGA platform for a given II. The first heuristics constrains the exploration space to significantly reduce the runtime, while achieving the same or even better results than the exact algorithm. Similarly, the second heuristics first reduces the exploration space, then groups the kernels with similar execution time on a single FPGA to minimize the FPGA frequency, thus minimizing the power consumption. In addition, it is thousands of times faster than the exact algorithm.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In chapter 3, we provide a new and fast method to maximize the throughput, i.e. minimize the initiation interval, of pipelined applications consisting of multiple kernels and deployed on multiple FPGAs. This method can efficiently optimize the number of parallel compute units of each kernel and allocate them on FPGAs in a way to respect the resource and memory bandwidth constraints. Since the method is based on an optimization problem that involves integer variables and is non-linear, it first has been solved using a Mixed-Integer Non-Linear Programming (MINLP) solver which, however, is highly inefficient. Due to this reason, we provided a heuristic method which includes a Geometric Programming (GP) solver and an allocator. The GP solver first relaxes the constraints and returns the number of Compute Units (CUs) of each kernel as a real number, but since we need an integer value we round it to both the smaller and greater nearest integer values, hence creating two possible alternative implementations for each kernel. Then, the allocator assigns the CUs on the FPGAs and selects the best implementation. By comparing the solutions obtained using MINLP solver and our heuristic method on a set of benchmarks, we could prove the effectiveness of our approach, which is much more efficient than the MINLP solver.

However, the first mathematical model implemented in our optimization framework did not include the data transfer time from the host CPU to the FPGA memory. Since the data transfer time can be very high for applications like CNNs since the data size of neuron activation is very high, neglecting the data transfer time could lead to suboptimal solutions. To improve the accuracy, a more detailed performance optimization model is presented in chapter 4. It includes the data transfer time. Moreover, it considers the frequency reduction introduced by the increasing resource usage of an FPGA. The execution time is divided into three parts, data transfer time from DDR to FPGA, computation time, and data transfer time from FPGA to DDR. We also separate the computation time and the data transfer time between DDR and FPGA. Similarly to the preliminary model, we first solved the optimization problem using a MINLP solver. However, due to the long CPU time and inefficiency of the solver, we propose a fast and accurate heuristic method. Experimental results show that our heuristic method can provide very similar results as the exact MINLP solution when the problem size is small, and it returns much better results for larger problem sizes.

We also compared the results obtained using these two models in chapter3 and chapter 4. In summary, the first “Simplified” model can be used for multi-kernel applications that do not have a big amount of data to transfer from one layer to the next layer. While the “Enhanced” model can be used for any multi-kernel application. They can provide a solution that maximizes the throughput of the multi-kernel application while respecting the resource constraints and it is highly efficient.

The second part of the thesis focuses on the power optimization model for multi-kernel applications targeting multi-FPGA platforms. We propose to upload at runtime the best power-optimized CNN implementation for a given throughput constraint. Our design method gives the best number of parallel instances of each kernel, their allocation to the FPGAs, the number of powered-on FPGAs and their clock frequency. We compared the proposed method with state-of-the-art methods including frequency scaling of the solution with the best throughput and replicating the solutions with the minimum throughput. Our method proved to be the best.

The power optimization problem has also been solved at first with a MINLP solver, which again proved to be highly inefficient. Thus we proposed two heuristic methods to efficiently obtain energy-optimal or near-optimal solutions to configure a multi-FPGA platform for a given II. The first heuristics constrains the exploration space to significantly reduce the runtime, then uses the same MINLP solver, which now can complete in a much shorter time. Similarly, the second heuristics first reduces the exploration space, then uses a greedy algorithm to further reduce the runtime. The results show that the first heuristic method can find better solution within the same time limit, while the second heuristic can return better solution and at the same time it is thousands of times faster than the exact algorithm.

7.2 Future Work

In the models that we proposed in the thesis, we did not consider direct on-chip data transfer from the previous layer to the next. All the data are transferred through DDR even when consecutive kernels are allocated on the same FPGA. This assumption is used to simplify the problem, because routing data between different CUs allocated on the same chip and on different chips would be very complex. However, it is redundant for kernels that are allocated on the same FPGA since they can exchange data directly on FPGA. In this way, it will reduce the bandwidth usage and reduce the total execution time.

For now the model can only be used for multi-kernel applications for which the computation workload can be split in multiple pipelined kernels, like CNNs. 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) can also be changed using the unrolling of loops inside each kernel, in addition to instantiating multiple CUs. This will provide a further level of optimization that can lead to better results.

There are some practical implementation details that we did not address. For example, how to allocate CUs on the multi-SLR FPGAs which state some constraints. Moreover, for now, we assume the DDR memory is uniform to simplify the model, but actually, the memory is banked over multiple DDR memory banks per FPGA. We will address these issues to make the model more precise.

Finally, in the power model, the data transfer from host CPU to FPGA DDR T_{h2f} and back T_{f2h} are done sequentially. However, the PCIe interface is duplex, the data transfer can be done in parallel to improve the performance.

Nomenclature

Acronyms / Abbreviations

2D-SPP	2D rectangular Strip-Packing Problem
AI	Artificial Intelligence
ALU	Algorithm-Logic Unit
ANN	Artificial Neural Network
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AWS	Amazon Web Service
AXI	Advanced eXtensible Interface
BRAM	Block RAM
CLB	Configurable Logic Block
CMOS	Complementary Metal–Oxide–Semiconductor
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DFE	Dataflow Engine
DL	Deep Learning
DNN	Deep Neural Network

DP	Dynamic Programming
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor
DUT	Device Under Test
EC2	Elastic Compute Cloud
EDA	Electronic Design Automation
FF	Flip-Flop
FIFO	First In First Out queue
FP16	Fixed-point 16 bits
FP32	Fixed-point 32 bits
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GFLOP	Giga Floating Point Operations Per Second
GP	Geometric Programming
GPU	Graphic Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
HPC	High Performance Computing
HW	Hardware
IDE	Integrated Development Environment
II	Initiation Interval
int8	Integer 8 bits
IP	Intellectual Property
LUT	Lookup Table
MINLP	Mixed-Integer Non-Linear Programming

ML	Machine Learning
NoC	Network On a Chip
OpenCL	Open Computing Language
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
REG	Register
RTL	Register Transfer Level
SDK	Software Development Kit
SLL	Super Long Line
SLR	Super Logic Region
SOC	Syetem On a Chip
SQL	Structured Query Language
SW	Software
TPU	Tensor Processing Unit
URAM	Unified-Random Access Memory
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLSI	Very Large-Scale Integration
XRT	Xilinx runtime

Bibliography

[1] Norman P Jouppi 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] P. Belotti. *Couenne (Convex Over and Under ENvelopes for Nonline Estimation)*. 2018. URL: https://www.coin-or.org/Couenne/ (visited on 11/26/2018).

[3] Matthias Koeppe. "On the Complexity of Nonlinear Mixed-Integer Optimization". In: 154 (June 2010). DOI: 10.1007/978-1-4614-1927-3_19.

[4] Stephen Boyd et al. "A tutorial on geometric programming". In: *Optimization and engineering* 8.1 (2007), p. 67.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[6] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[7] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2016). DOI: 10.1109/cvpr.2016.90. URL: http://dx.doi.org/10.1109/CVPR.2016.90.

[8] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[9] Hadi Esmaeilzadeh et al. "Power Challenges May End the Multicore Era". In: *Commun. ACM* 56.2 (Feb. 2013), pp. 93–102. ISSN: 0001-0782. DOI: 10.1145/2408776.2408797. URL: https://doi.org/10.1145/2408776.2408797.

[10] Ding Xie, Jimmei Lai, and Jiarong Tong. "A high utilization rate routing algorithm for modern FPGA". In: *2008 9th International Conference on Solid-State and Integrated-Circuit Technology*. 2008, pp. 2333–2336. DOI: 10.1109/ICSICT.2008.4735047.

[11] Michael Fingeroff. *High-level synthesis: blue book*. 2010.

[12] *SDAccel Development Environment*. en. URL: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html (visited on 02/16/2020).

[13] Michael I. Gordon, William Thies, and Saman Amarasinghe. "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs". In: *SIGOPS Oper. Syst. Rev.* 40.5 (Oct. 2006), pp. 151–162.

[14] Sander Stuijk, Marc Geilen, and Twan Basten. "Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs". In: *Computers, IEEE Transactions on* 57 (Nov. 2008), pp. 1331–1345.

[15] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. "Synergistic Execution of Stream Programs on Multicores with Accelerators". In: *Proceedings of ACM SIGPLAN/SIGBED 2009*. 2009, pp. 99–108.

[16] Yongming Shen, Michael Ferdman, and Peter Milder. "Maximizing CNN Accelerator Efficiency Through Resource Partitioning". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 535–547. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080221.

[17] K. Guo et al. "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. DOI: 10.1109/ISVLSI.2016.129.

[18] Xiaofan Zhang et al. "DNNBuilder: An Automated Tool for Building High-performance DNN Hardware Accelerators for FPGAs". In: *Proceedings of the International Conference on Computer-Aided Design*. ICCAD '18. San Diego, California: ACM, 2018, 56:1–56:8. ISBN: 978-1-4503-5950-4. DOI: 10.1145/3240765.3240801.

[19] Jason Cong, Muhuan Huang, and Peng Zhang. "Combining Computation and Communication Optimizations in System Synthesis for Streaming Applications". In: *Proceedings of the 2014 ACM/SIGDA International Symposium on FPGAs*. 2014, pp. 213–222.

[20] Weiwen Jiang et al. "Achieving Super-Linear Speedup across Multi-FPGA for Real-Time DNN Inference". In: *ACM Transactions on Embedded Computing Systems* 18.5s (Oct. 2019), pp. 1–23. ISSN: 1539-9087. DOI: 10.1145/3358192.

[21] Eric Chung et al. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave". In: *IEEE Micro* 38 (Mar. 2018), pp. 8–20. URL: https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/.

[22] Jeremy Fowers et al. "A Configurable Cloud-Scale DNN Processor for Real-Time AI". In: *Proceedings of the 45th International Symposium on Computer Architecture, 2018*. ACM, June 2018. URL: https://www.microsoft.com/en-us/research/publication/a-configurable-cloud-scale-dnn-processor-for-real-time-ai/.

[23] Chen Zhang et al. "Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster". In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ISLPED '16. San Francisco Airport, CA, USA: Association for Computing Machinery, 2016, pp. 326–331. ISBN: 9781450341851. DOI: 10.1145/2934583.2934644. URL: https://doi.org/10.1145/2934583.2934644.

[24] W. Zhang et al. "An Efficient Mapping Approach to Large-Scale DNNs on Multi-FPGA Architectures". In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2019, pp. 1241–1244. DOI: 10.23919/DATE.2019.8715174.

[25] Luca Piccolboni et al. "COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators". In: *ACM Trans. Embed. Comput. Syst.* 16.5s (Sept. 2017).

[26] W. Jiang et al. "Heterogeneous FPGA-Based Cost-Optimal Design for Timing-Constrained CNNs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (Nov. 2018), pp. 2542–2554.

[27] J. Shen et al. "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.

[28] Junzhong Shen et al. "Accelerating 3D CNN-based Lung Nodule Segmentation on a Multi-FPGA System". In: *FPGA*. 2019.

[29] C. Kritikakis and D. Koch. "End-to-end Dynamic Stream Processing on Maxeler HLS Platforms". In: *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Vol. 2160-052X. 2019, pp. 59–66. DOI: 10.1109/ASAP.2019.00-29.

[30] J. Salom and H. A. Fujii. "Selected HPC Solutions Based on the Maxeler DataFlow Arrpoach". In: 2013.

[31] Andreas Bortfeldt. "A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces". In: *European Journal of Operational Research* 172.3 (2006), pp. 814–837. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2004.11.016. URL: https://www.sciencedirect.com/science/article/pii/S0377221704008598.

[32] L. Chen, T. Marconi, and T. Mitra. "Online scheduling for multi-core shared reconfigurable fabric". In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2012, pp. 582–585. DOI: 10.1109/DATE.2012.6176537.

[33] N. Ntene and J.H. van Vuuren. "A survey and comparison of guillotine heuristics for the 2D oriented offline strip packing problem". In: *Discrete Optimization* 6.2 (2009), pp. 174–188. ISSN: 1572-5286. DOI: https://doi.org/10.1016/j.disopt.2008.11.002. URL: https://www.sciencedirect.com/science/article/pii/S1572528608000844.

[34] A. Sailer et al. "Optimizing the task allocation step for multi-core processors within AUTOSAR". In: *2013 International Conference on Applied Electronics*. 2013, pp. 1–6.

[35] A. Sailer et al. "Optimizing the task allocation step for multi-core processors within AUTOSAR". In: *2013 International Conference on Applied Electronics*. 2013, pp. 1–6.

[36] O. Melnikova, I. Hahanova, and K. Mostovaya. "Using multi-FPGA systems for ASIC prototyping". In: *2009 10th International Conference - The Experience of Designing and Application of CAD Systems in Microelectronics*. 2009, pp. 237–239.

[37] Sameh Asaad et al. "A Cycle-Accurate, Cycle-Reproducible Multi-FPGA System for Accelerating Multi-Core Processor Simulation". In: FPGA '12. Monterey, California, USA: Association for Computing Machinery, 2012, pp. 153–162. ISBN: 9781450311557. DOI: 10.1145/2145694.2145720. URL: https://doi.org/10.1145/2145694.2145720.

[38] William N.N. Hung and Richard Sun. "Challenges in Large FPGA-Based Logic Emulation Systems". In: *Proceedings of the 2018 International Symposium on Physical Design*. ISPD '18. Monterey, California, USA: Association for Computing Machinery, 2018, pp. 26–33. ISBN: 9781450356268. DOI: 10.1145/3177540.3177542. URL: https://doi.org/10.1145/3177540.3177542.

[39] Kouadri-Mostefaoui et al. "Large Scale On-Chip Networks : An Accurate Multi-FPGA Emulation Platform". In: *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*. 2008, pp. 3–9. DOI: 10.1109/DSD.2008.130.

[40] X. Li and O. Hammami. "Multi-FPGA emulation of a 48-cores multiprocessor with NOC". In: *2008 3rd International Design and Test Workshop*. 2008, pp. 205–208. DOI: 10.1109/IDT.2008.4802498.

[41] G. Karypis et al. “Multilevel hypergraph partitioning: applications in VLSI domain”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pp. 69–79. DOI: 10.1109/92.748202.

[42] S. Zhao et al. “A universal self-calibrating Dynamic Voltage and Frequency Scaling (DVFS) scheme with thermal compensation for energy savings in FPGAs”. In: *2016 IEEE Applied Power Electronics Conference and Exposition (APEC)*. 2016, pp. 1882–1887. DOI: 10.1109/APEC.2016.7468125.

[43] J. Luis Nunez-Yanez, M. Hosseinabady, and A. Beldachi. “Energy Optimization in Commercial FPGAs with Voltage, Frequency and Logic Scaling”. In: *IEEE Transactions on Computers* 65.5 (2016), pp. 1484–1493. DOI: 10.1109/TC.2015.2435771.

[44] Joshua M. Levine, Edward Stott, and Peter Y.K. Cheung. “Dynamic Voltage amp; Frequency Scaling with Online Slack Measurement”. In: *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’14. Monterey, California, USA: Association for Computing Machinery, 2014, pp. 65–74. ISBN: 9781450326711. DOI: 10.1145/2554688.2554784. URL: https://doi.org/10.1145/2554688.2554784.

[45] Selome Kostentions Tesfatsion et al. “Power and performance optimization in FPGA-accelerated clouds”. In: *Concurrency and Computation: Practice and Experience* 30.18 (2018), e4526.

[46] Chen Zhang et al. “Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster”. In: *Proc. 2016 Int. Symp. on Low Power Electronics and Design*. ISLPED ’16. San Francisco Airport, CA, USA: ACM, 2016, pp. 326–331. ISBN: 978-1-4503-4185-1. DOI: 10.1145/2934583.2934644.

[47] Naif Tarafdar et al. “Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center”. In: *Proc. 2017 ACM/SIGDA Int. Symp. on FPGAs*. Monterey, California, USA: ACM, 2017, pp. 237–246. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021742.

[48] A. M. Caulfield et al. “A cloud-scale acceleration architecture”. In: *49th IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783710.

[49] Bingbing Li et al. “FTRANS: Energy-Efficient Acceleration of Transformers Using FPGA”. In: *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED ’20. Boston, Massachusetts: Association for Computing Machinery, 2020, pp. 175–180. ISBN: 9781450370530. DOI: 10.1145/3370748.3406567.

[50] Junnan Shan et al. "Exact and Heuristic Allocation of Multi-Kernel Applications to Multi-FPGA Platforms". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC '19. Las Vegas, NV, USA: Association for Computing Machinery, 2019. ISBN: 9781450367257. DOI: 10.1145/3316781.3317821.

[51] Chen 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*. FPGA '15. Monterey, California, USA: ACM, 2015, pp. 161–170. ISBN: 978-1-4503-3315-3.

[52] Jan Kronqvist et al. "A review and comparison of solvers for convex MINLP". In: *Optimization and Engineering* 20.2 (2019), pp. 397–455.

[53] Omprakash K. Gupta and A. Ravindran. "Branch and Bound Experiments in Convex Nonlinear Integer Programming". In: *Management Science* 31.12 (1985), pp. 1533–1546. ISSN: 00251909, 15265501. URL: http://www.jstor.org/stable/2631793.

[54] A. Sayed, A. N. Mohieldin, and M. Mahroos. "A Fast and Accurate Geometric Programming Technique for Analog Circuits Sizing". In: *2019 31st International Conference on Microelectronics (ICM)*. 2019, pp. 316–319. DOI: 10.1109/ICM48031.2019.9021474.

[55] E. Burnell and W. Hoburg. *GPkit*. 2018. URL: https://github.com/convexopt/gpkit (visited on 11/26/2018).

[56] J. Shan et al. "CNN-on-AWS: Efficient Allocation of Multi-Kernel Applications on Multi-FPGA Platforms". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020), pp. 1–1. DOI: 10.1109/TCAD.2020.2994256.

[57] Chen 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*. FPGA '15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170. ISBN: 9781450333153. DOI: 10.1145/2684746.2689060. URL: https://doi.org/10.1145/2684746.2689060.

[58] Jonathan Pedoeem and Rachel Huang. "YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers". In: *CoRR* abs/1811.05588 (2018). arXiv: 1811.05588. URL: http://arxiv.org/abs/1811.05588.

[59] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[60] J. Shan et al. "Power-Optimal Mapping of CNN Applications to Cloud-Based Multi-FPGA Platforms". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.12 (2020), pp. 3073–3077. DOI: 10.1109/TCSII.2020.2998284.

[61] *Power Calculators*. en. URL: https://www.micron.com/support/tools-and-utilities/power-calc (visited on 02/16/2020).

[62] *Xilinx Power Estimator (XPE)*. en. URL: https://www.xilinx.com/products/technology/power/xpe.html (visited on 02/16/2020).

This Ph.D. thesis has been typeset by means of the TeX-system facilities. The typesetting engine was pdfLaTeX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TeX-system installation.