University of Bremen

University of Bremen
Faculty 3 - Mathematics and Computer Science

Doctoral thesis

submitted to obtain the degree of
Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

Towards Sustainable Artificial Intelligence Systems: Enhanced System Design with Machine Learning based Design Techniques

Author:	Christopher A. Metz
	Email: cmetz@uni-bremen.de
First Examiner:	Prof. Rolf Drechsler
	Email: drechsler@uni-bremen.de
Second Examiner:	Prof. Görschwin Fey
	Email: goerschwin.fey@tuhh.de

Date of Kolloquium: 29.05.2024

Acknowledgment

I would like to express my sincere gratitude to Rolf Drechsler for his invaluable guidance and support throughout my doctoral thesis journey. His expertise, patience, and dedication have been instrumental in shaping my research and helping me overcome the challenges. I would like to thank Görschwin Fey for serving as the second examiner.

I also thank Lena Steinman and the whole Data Science Center team for providing me with the resources and infrastructure necessary for conducting my research. The collaborative environment and cutting-edge facilities were crucial in facilitating my work.

Furthermore, I am grateful to my colleagues who stood by me, offering insights, feedback, and encouragement during this challenging process. Your camaraderie and shared experiences made the academic endeavor more rewarding and enjoyable.

Special thanks go to Mehran Goli, who supported me during the early phases of this thesis and always assisted me with advice on any questions I had. Moreover, I'd like to thank the proofreaders Sascha Haverland, Muhammad Hassan, and Sarah Koopmann, who spent many hours.

Last, I'd like to thank my love, Sarah Koopmann, and my family for their inexhaustible support. This achievement was only possible with the combined efforts of everyone involved. Thank you for participating in this significant milestone in my academic journey. Your support has been truly invaluable.

Abstract

Efficient and timely calculations of Machine Learning (ML) algorithms are crucial for emerging technologies like autonomous driving, the Internet of Things (IoT), and edge computing. One of the primary ML techniques used in such systems is Convolutional Neural Networks (CNNs), which demand high computational resources. This requirement has led to using ML accelerators like General Purpose Graphig Processing Units (GPGPUs) to meet design constraints. However, GPGPUs have high power consumption needs, and thus, selecting the most suitable accelerator involves Design Space Exploration (DSE). This process is usually time-consuming and requires significant manual effort. This thesis presents approaches to improve the DSE process by supporting the identification of the most appropriate GPGPU for CNN inferencing systems. Different techniques are developed to quickly and precisely forecast the power consumption and performance of CNNs during inference. These approaches empower the system designer to estimate power consumption and performance for GPGPUs in the early stages of development without executing the application on real devices. Without the need to execute and profile applications on real devices, the number of prototypes can significantly be reduced.

Besides the system's power and performance requirements and the ML accelerator selection, the designer has to face the placement problem and decide whether an application is implemented on an IoT device or in the Cloud. The available network, bandwidth, and latency are crucial if the application is implemented in the Cloud. Therefore, this thesis presents a decision-supporting system that is pivotal in helping system designers Make these complex decisions. This system is designed to consider the available network, bandwidth, and latency, and it distinguishes between power or performance optimization needs, thereby empowering the system designer to make informed choices.

Zusammenfassung

Effiziente und zeitnahe Berechnungen von Algorithmen des maschinellen Lernens (ML) sind für neue Technologien wie autonomes Fahren, das Internet der Dinge (engl. Internet of Things (IoT)) und Edge Computing entscheidend. Eine der wichtigsten ML-Techniken, die in solchen Systemen verwendet werden, sind Convolutional Neural Networks (CNN), die hohe Rechenressourcen erfordern. Diese Anforderung hat dazu geführt, dass ML-Beschleuniger wie General Purpose Graphig Processing Units (GPGPUs) eingesetzt werden, um die Designvorgaben zu erfüllen. GPGPUs haben jedoch einen hohen Stromverbrauch, so dass die Auswahl des am besten geeigneten Beschleunigers eine aufwendige, manuelle Durchsuchung des Designraumes erfordert (engl. Design Space Exploration (DSE)).

Diese Arbeit stellt Ansätze zur Verbesserung des DSE-Prozesses vor. Es werden verschiedene Techniken entwickelt, um den Stromverbrauch und die Leistung von CNNs während der Ausführung vorherzusagen. Das ermöglichen es dem Systementwickler, den Stromverbrauch und die Leistung von GPGPUs in den frühen Phasen der Entwicklung abzuschätzen, ohne die Anwendung auf realen Geräten ausführen zu müssen. Dadurch kann die Anzahl der Prototypen erheblich reduziert werden. Zusätzlich muss sich der Entwickler mit dem Problem der Platzierung auseinandersetzen und entscheiden, ob eine Anwendung auf einem IoT-Gerät oder in der Cloud implementiert wird. Das verfügbare Netzwerk, die Bandbreite und die Latenzzeit sind entscheidend, wenn die Anwendung in der Cloud implementiert wird. Daher wird in dieser Doktorarbeit ein System vorgestellt, das Entwicklern bei dieser komplexen Entscheidungen hilft. Das System ist so konzipiert, dass es das verfügbare Netzwerk, die Bandbreite und die Latenzzeit berücksichtigt, wodurch der Systementwickler, unterberücksichtigung von Energie- und Leistungsanforderungen, eine Entscheidungen zu treffen kann.

Contents

Acknowledgment III

Abstact V

Zusammenfassung VII

List of Figures XIV

List of Tables XV

Acronyms XVII

1 Introduction 1
- 1.1 Impact on Structural Characteristics of Data Centers . . . 6
- 1.2 Challenges on Edge and IoT System Design for AI . . . 9
- 1.3 State-of-the-Art Solutions . . . 11
  - 1.3.1 Power and Performance Prediction . . . 11
  - 1.3.2 Offloading . . . 13
  - 1.3.3 Limitation of State-of-the-Art approaches . . . 14
- 1.4 Contributions . . . 16

2 Background 23
- 2.1 Energy Consumption Metrics . . . 23
- 2.2 Neural Networks . . . 24
  - 2.2.1 Convolutional Neural Networks . . . 25
  - 2.2.2 Benchmark CNNs . . . 26
- 2.3 Machine Learning Exchange Formats . . . 27
  - 2.3.1 Open Neural Network Exchange . . . 27
  - 2.3.2 Save and Load TensorFlow Models . . . 29
- 2.4 Graphic Processing Units . . . 29

3 Performance and Energy Trade-off Analysis of CNN Offloading 33
- 3.1 Introduction . . . 34
- 3.2 Methodology . . . 36
  - 3.2.1 Model Description Schema . . . 36

3.2.2 Statistical Modeling . . . 38
3.3 Experimental Results . . . 42
3.3.1 Evaluation Setup . . . 42
3.3.2 Result Description . . . 46
3.4 Discussion . . . 56
3.4.1 Evaluation interpretation . . . 57
3.4.2 Threats to validity . . . 59
3.5 Conclusion . . . 60

4 Power Estimation of Convolutional Neural Networks on GPGPUs 61
4.1 Introduction . . . 61
4.2 Methodology . . . 63
4.2.1 GPGPUs Architecture Analysis . . . 64
4.2.2 PTX Instructions Analysis . . . 65
4.2.3 CNN Topology Analysis . . . 66
4.2.4 Creating Training Dataset and Predictive Model . . . 67
4.3 Experimental Results . . . 71
4.4 Discussion . . . 74
4.5 Conclusion . . . 76

5 Power and Performance Analysis with Dynamic Frequency Scaling 79
5.1 Introduction . . . 80
5.2 Methodology . . . 82
5.2.1 Training Data Generation . . . 82
5.2.2 Differentiation of Methodologies . . . 85
5.2.3 Performance Evaluation Process . . . 85
5.2.4 Predictive Model Generation and Evaluation . . . 86
5.3 Experimental Results . . . 87
5.3.1 Performance Behavior Analysis . . . 88
5.3.2 Power consumption Modeling . . . 93
5.4 Discussion . . . 97
5.4.1 Performance Evaluation Discussion . . . 97
5.4.2 Predictive Model Discussion . . . 98
5.5 Conclusion . . . 99

6 Hybrid PTX Analysis 101
6.1 Introduction . . . 101
6.2 Methodology . . . 104
6.2.1 Running Example . . . 104
6.2.2 Static PTX Analysis . . . 105
6.2.3 Dynamic PTX Analyzation . . . 108

6.2.4 Profile Generation . . . 112
6.3 Experimental Results . . . 112
6.4 Discussion . . . 116
6.4.1 Evaluation interpretation . . . 117
6.4.2 Limitations . . . 118
6.5 Conclusion . . . 118

7 Performance Estimation of CNNs for GPGPUs 121
7.1 Introduction . . . 121
7.2 Methodology . . . 123
7.2.1 Training Dataset Creation . . . 123
7.2.2 Predictive Model Generation and Evaluation . . . 126
7.2.3 Differentiation of Methodologies . . . 127
7.3 Experimental Results . . . 127
7.4 Discussion . . . 134
7.5 Conclusion . . . 134

8 Discussion 135

9 Conclusion 139

References XXI

List of Figures

1.1 Overview of IoT systems; adapted from [1]. 2
1.2 Comparision of maximum power consumption of all current 14 data center GPGPUs from NVIDIA. 5
1.3 Comparing different neural network engine implementations on NVIDIA Jeston Nano 4GB. 10
1.4 Improving the DSE analysis time by combining HyPA for data gathering and predictive models for performance estimation. 18

2.1 The general architecture of a CNN model. 26
2.2 Overview of Streaming Multiprocessor architecture 30
2.3 Example PTX file. 31

3.1 Overview of the functionality of MoReA core and HTTP service. . . 37
3.2 General model description structure for ML-model description . . . 38
3.3 Experimental setup for MoReA 43
3.4 Depiction variation coefficients of energy consumption for all setups. 46
3.5 Depiction variation coefficients of energy consumption for all setups. 47
3.6 Depiction of average energy consumption for local execution with TensorFlow and ONNX formats differentiated by phases. Boxplots represent the CNN results. 48
3.7 Depiction of average computation time for local execution with ONNX and remote execution with different connectivities. Boxplots represent the CNN results. 50
3.8 Depiction of average energy consumption for local execution with ONNX and remote execution with different connectivities. Boxplots represent the CNN results. 51
3.9 Depiction of total average energy consumption and total computation time per CNN, differentiated by evaluation setup. 52
3.10 Depiction of the number of trainable weights of the CNN with energy consumption for SETUP- and PROCESSING-phases for different evaluation setups. 53
3.11 Depiction of the number of trainable weights of the CNN with the energy consumption ratio to image size for SETUP-phase for the local setup with ONNX format. 54

3.12 Depiction of the number of trainable weights of the CNN with the energy consumption ratio to image size for PROCESSING-phases for the local setup with ONNX format. 55
3.13 Number of won hypothesis tests by remote setup connectivity. . . . 56
3.14 Depiction of t-statistic value for individual CNNs against their weights, colors depicting different connectivities. 57

4.1 Overview of the proposed methodology. 65
4.2 MAPE value for different k ranging from one to 20. 71
4.3 Scatter plot of the predicted power consumption for different CNNs on the NVIDIA GTX 1080 Ti with 8GB Memory. 74
4.4 AE of power estimation of different CNNs for NVIDIA RTX 1080Ti. . 75

5.1 DFS-based power estimation methodology. 83
5.2 Part of SLURM SBATCH Job script to execute the CNN benchmarks with different frequencies on the experimental setup. 84
5.3 Average computation time (in seconds) for all CNNs and frequencies with error bands of one σ 89
5.4 Average maximum power (in W) for all CNNs and frequencies with error bands of one σ 90
5.5 Relative average computation time for all CNNs and frequencies with a loess-model fitted 92
5.6 Correlation coefficients for frequency and computation time as well as frequency and power 94
5.7 Comparison of predicted and real power consumption for six different CNNs for frequencies between 397 MHz and 1590 MHz on the NVIDIA V100S GPGPU. 96

6.1 General workflow of the PTX analyzer HyPA 105
6.2 Running example of PTX to illustrate the workflow of HyPA 106
6.3 Run-time comparison for HyPA and nvprof 114
6.4 Run-time comparison for HyPa and nvprof in correlation 116

7.1 Performance estimation methodology. 124
7.2 Decision Tree predicted 130
7.3 KNN predicted 130
7.4 Gradient Boosting predicted 131
7.5 Random Forest Tree predicted 131
7.6 Short snippet from the generated decision tree 133

List of Tables

2.1 An overview of CNN models used in the experiments 28

3.1 CNN models and parameters for study (a subset of the CNNs in tab. 2.1 in chapter 2). 44
3.2 Bandwidth and latency settings. 45
3.3 Summarised decision support 59

4.1 Example of training dataset structure used to create the predictive model 68
4.2 Experimental results for different input feature combinations 73
4.3 Descriptions of the best predictors (features) combination 73

5.1 Comparison of four different ML-regression algorithms in terms of accuracy and execution time 95

6.1 Experimental results for static and dynamic analysis with HyPA for 32 different CNNs. 120

7.1 Comparison of four different ML-regression algorithms in terms of accuracy and execution time 127
7.2 Predictors descriptions used by the decision tree 128
7.3 Execution time comparison of the proposed approach versus the naive approach for eight different CNNs in the case of various GPGPUs 132

Acronyms

AI	Artificial Intelligence
APE	Absolute Percentage Error
AE	Absolute Error
API	Application Programming Interface
CNN	Convolutional Neural Network
CPS	Cyber-Physical-Systems
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CTA	Cooporate Thread Array
CFIL	Control Flow Instruction List
CSV	Comma-Separated Values
DSE	Design Space Exploration
DNN	Deep Neural Network
DFS	Dynamic Frequency Scaling
DVFS	Dynamic Voltage and Frequency Scaling
DL	Deep Learning
EECOF	Energy Efficient Computational Offloading Framework
ERP	Enterprise Resource Planning
FDG	Filtered Dependency Graph
FP	Floating Point

FPGA	Field Programmable Gate Array
GPGPU	General Purpose Computing on Graphic Processing Unit
GFLOPs	Giga Floating Point Operations per Second
GPT	Generative pre-trained transformers
HTTP	Hyper Text Transport Protocol
HyPA	Hybrid PTX Analysis
HPC	High-Performance Computing
IoT	Internet of Things
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
IPC	Instruction per Cycle
K-NN	K-Nearest Neighbor
kWh	kilowatt hour
LLVM	Low Level Virtual Machine
LLM	Large Language Model
LoRaWAN	Long Range Wide Area Network
ML	Machine Learning
MAPE	Mean Absolute Percentage Error
MoReA	Model Request API
MDS	Model Description Schema
MDF	Model Description File
MCC	Mobile Cloud Computing
MWh	megawatt hour
NAS	Neural Architecture Search

NN	Neural Network
nvcc	NVIDIA CUDA Compiler
NHS	Neural Hardware Search
NLP	Natural Language Processing
ONNX	Open Neural Network Exchange
PTX	Parallel Thread Execution
PU	Processing Unit
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
RMSE	Root Mean Squared Error
RNN	Rekurrent Neural Network
ReLU	Rectified Linear Unit
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RFT	Random Forest Tree
SoC	System on a Chip
SoM	System-on-a-Module
SSA	Static Single Assignment
SLURM	Simple Linux Ressource Manager
SM	Streaming Multiprocessor
SIMT	Single Instruction Multiple Thread
SIMD	Single Instruction Multiple Data
SaaS	Software as a Service

SOA	Service Oriented Architecture
TDP	Thermal Design Power
TWh	terawatt hour
UAV	Unmanned Arial Vehicle
VM	Virtual Machine
XGBoost	eXtreme Gradient Boosting

1 Introduction

The manufacturing industry produces a giant amount of data from various sources, including sensors on production lines, environmental data, machine tool parameters, and Enterprise Resource Planning (ERP) systems [2]. The growing volume of data can be attributed to the advancements in Internet of Things (IoT), which aim to connect the physical world with the virtual world. This requires many sensors and devices of all types. Some refer to these connected systems as Cyber-Physical-Systems (CPS) [1].

The manufacturing industry can maximize its potential by connecting physical devices and collecting sensor data. For instance, utilizing the extensive amount of information, often referred to as *Big Data* [2], can greatly aid in optimizing manufacturing operations by implementing predictive maintenance [3] and lean management [4] techniques. However, with big data come new challenges, such as data heterogeneity, formats, semantics, and quality. These challenges can cause distractions from the main issues and lead to incorrect conclusions. Therefore, new automated methods for data analysis are necessary. With the latest advancements in mathematics and computer science and the availability of user-friendly and free frameworks, big data can be optimally utilized. As a result, the most recent developments in Machine Learning (ML) are being incorporated into the latest IoT and Edge advancements to effectively manage the manufacturing industry's big data, leading to a significant rise in the use of ML applications [2].

Fig. 1.1 illustrates IoT systems' intricate and diverse nature. The top level is consistently linked to the cloud through different network types, leading to the Fog, Edge, or IoT device. Nevertheless, certain IoT devices require gateways, such as brokers or radio communication basis stations, to establish a connection. A significant research area in these IoT systems is the *placement of functions*, as these IoT and Edge devices often face limitations in terms of power, performance,

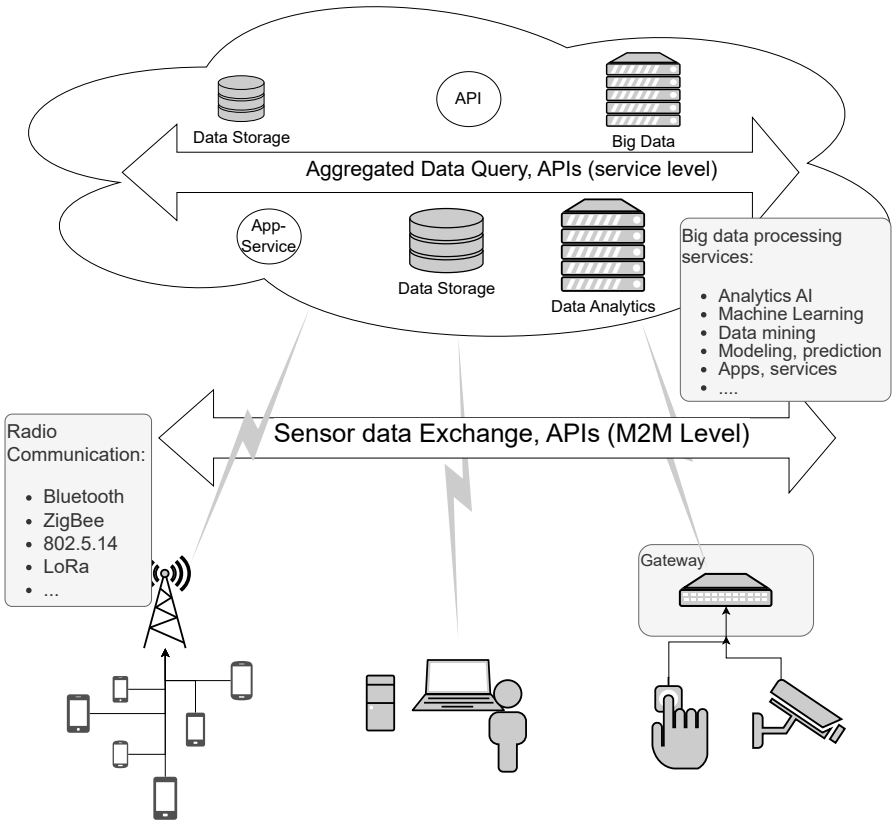


Figure 1.1: Overview of IoT systems; adapted from [1].

bandwidth, and so on [1]. To determine the best location for computational workloads, system designers must evaluate the power consumption, performance, and bandwidth requirements for different devices to decide if the computation is getting to the data or the data to the computation resources [1]. Unfortunately, this process is often done manually and involves a lot of time and cost, especially if multiple prototypes need to be tested due to the vast design space. Thus, new challenges in Design Space Exploration (DSE) arise for designing ML-based IoT and Edge devices as well as cloud services [5, 6, 7, 8, 9].

For instance, assume an IoT application constantly streaming a video and aiming to detect humans on the stream. In such a case, only those images or video snippets detecting a human are of interest. Thus, offloading the Artificial Intelligence (AI)-based object detection to the cloud and forwarding the complete video

stream would stress the network and cause a lot of unneeded data in the cloud systems. Consequently, to only forward essential data to the cloud for Big Data analysis, AI-based object detection should be moved to the Edge [1].

One of the most popular techniques for object detection is Convolutional Neural Network (CNN). They can be used for various image processing tasks such as image classification, object detection, or motion recognition [10]. However, CNNs require high computational resources. The convolutional layers within a CNN are responsible for over 90 % of the computation [11]. While larger IoT and Edge devices can handle these heavy calculations, smaller devices (e.g., sensors) may require additional computing units for local data processing or offloading the computation to the cloud. Modern processing units and techniques like General Purpose Computing on Graphic Processing Unit (GPGPU)s are one solution to handle such computational-intensive workloads.

GPGPUs (e.g., NVIDIA V100) can offer 32-times better performance on Deep Neural Networks (DNNs) than Central Processing Units (CPUs) [12]. For instance, by utilizing the parallelism of 256 GPGPUs, the Resnet50 [13], a CNN with 50 hidden layers, can be trained on the complete ImageNet [14] dataset in just one hour [15]. In contrast, when the Resnet50 was first introduced by [13], it took 29 hours to train [16].

The current state of industry and academia has shown that GPGPUs have become a dominant force in accelerating the inference and training of DNNs. This trend is notable in the widespread adoption of GPGPUs in various systems and applications. For instance, in June 2015, nearly 19 % of the systems on the TOP500 list used additional GPGPU accelerators, but this proportion increased to almost 30 % in June 2022 [17]. The use of GPGPUs for training and inferencing of DNN is a significant reason for this increase [18, 19]. This development indicates the need for a new design paradigm for systems to handle ML workloads.

Besides all their apparent merits, GPGPUs' drawback is their high power consumption to achieve their high-performance levels. The latest developments include GPGPUs explicitly designed for ML training and inference, which can consume up to 700 watts per GPGPU. Fig. 1.2 illustrates the high variety of GPGPUs' power consumption ranging from 70 watts to 700 watts for the latest NVIDIA data center GPGPUs. As most AI systems have multiple GPGPUs per machine

[20], the power consumption of ML and AI systems presents new challenges [5, 6, 7, 8, 9, 21].

An example of extreme power consumption in data centers can be seen in the Summit, a supercomputer located at the Oak Ridge National Laboratory, Tennessee, USA, with 27,648 NVIDIA Volta GPGPUs that consume 13 million watts [19]. However, by implementing power savings of 5 %, significant yearly cost savings of up to 1 million dollars can be achieved [22]. Unfortunately, High-Performance Computing (HPC) data centers do not support relocating applications to greener locations as cloud providers do. Thus, energy reduction can only be implemented by changing the AI model and accelerator (i.e., GPGPU) or making the costly and intensive switch to the cloud [23]. On the other hand, smaller IoT devices can also experience increased power consumption due to ML inferencing. For instance, executing object recognition on an NVIDIA Jetson TX1 can consume 7 watts, but offloading the same task to the cloud reduces power consumption to 2 watts [24]. Offloading ML tasks of IoT applications with limited battery resources can be a promising strategy. Using cloud infrastructure for AI applications can lead to an energy cost reduction by a factor of 1.4 - 2 [23]. Consequently, computing on cloud infrastructure increases sustainability and reduces the carbon footprint of AI applications.

Additionally, the feasibility of offloading ML workloads depends on available bandwidth, system architecture, and data flow of the IoT environment [1]. Therefore, local execution may be necessary when offloading is not viable. However, offloading ML tasks to the cloud does not reduce energy consumption. While the IoT device needs less power, additional power consumption in the cloud is generated. Thus, the overall energy consumption might increase as more powerful cloud machines run the calculations and the energy consumption of the IoT device waits for the calculations' results. Consequently, offloading moves the energy consumption to the cloud.

The range of IoT devices and the number of available GPGPUs are extensive (see Fig. 1.1). In addition, Edge devices can vary in size and complexity, ranging from small embedded systems with microprocessors over System on a Chip (SoC) to larger multicore systems with gigabytes of memory [1]. As the computer architecture design space is vast, automatic system design and exploration

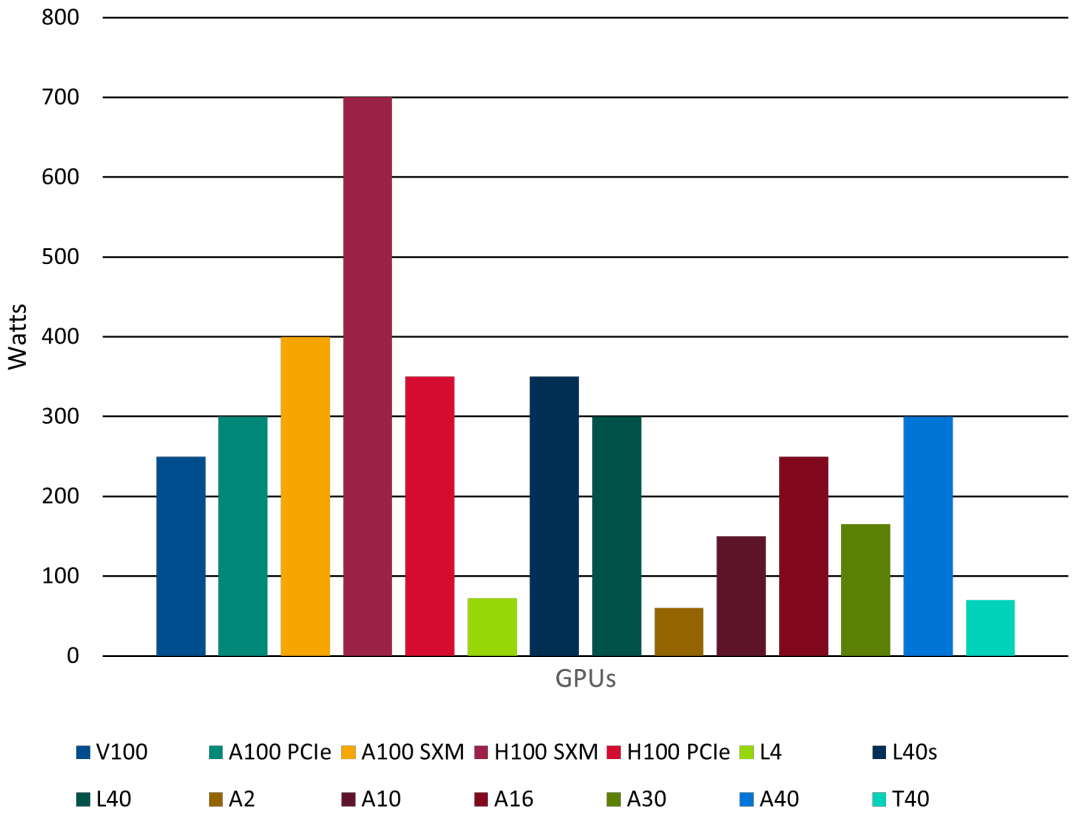


Figure 1.2: Comparision of maximum power consumption of all current 14 data center GPGPUs from NVIDIA.

approaches are necessary to meet sustainable requirements and time-to-market. Based on the *4Ms: model, machine, mechanization, and map*, different practices can be pursued to gain sustainability in reducing energy consumption and carbon emissions [23]. The *4Ms* are defined as follows:

1. **Model:** The selected ML model has impact on the energy consumption. By selecting an efficient ML model architecture, the required computations can be reduced by a factor of 5-10 [23].

2. **Machine:** Choosing an appropriate accelerator (e.g., GPGPU) for ML training can improve the performance/watt by a factor of 2-5 [23].

3. **Mechanization:** Computing in the cloud reduces the energy cost by a factor of 1.4-2 compared to on-premise data centers. As cloud data centers

are designed for energy efficiency, they usually outperform on-premise data centers in smaller, older spaces not designed to be energy-efficient [23].

4. **Map:** Cloud provider offer their customers to pick the location (e.g., data center) for their application. By this, the cloud customer can pick an energy-efficient data center with high portions of renewable energy. Consequently, it further reduces the carbon footprint of the applications running in the cloud [23].

When designing an AI-based system for a given AI model, designers face the placement problem (i.e., machine, mechanization, and map). They must decide where computationally intensive applications should be executed. Hence, it is of utmost importance to answer the following challenge in the early design phases of new AI-based systems to ensure a sustainable system:

Challenge 1. ***Placement problem:*** *System designers must decide whether to run AI applications in the cloud, Edge, or offline on special devices. To make this decision, various factors like bandwidth, network latency, or the size and type of AI application must be considered. Without a decision-supporting system, it is difficult to decide the application placement and requires lots of tests.*

1.1 Impact on Structural Characteristics of Data Centers

Due to offloading strategies, more and more AI and ML applications are executed in data centers where powerful machines are available to handle computationally intensive tasks for training and inferencing. Consequently, the power and energy consumption of the selected data center is increasing. The increasing energy consumption has consequences on the structural characteristics of data centers due to laws (e.g., German energy-efficient law), sustainable aspects, or rising energy costs. As AI optimized machines are equipped with multiple GPGPUs, the power consumption and Thermal Design Power (TDP) place special requirements on data centers. For instance, to provide ChatGPT, a total of 3,617 NVIDIA

HGX A100 servers, equipped with 28,936 GPGPUs, are required. This amount of servers, and especially GPGPUs, lead to a daily energy consumption of 564 megawatt hour (MWh) [25]. For instance, if such a system were physically located in Germany, the cost of running it would be enormous. The energy price in Germany was $0.53 per kilowatt hour (kWh) in 2022 [26], resulting in a daily energy cost of $298,920 in Germany.

$$564,000kWh \cdot 0.53 \frac{\$}{kWh} = \$298,920 \tag{1.1}$$

Compared to the US with lower energy costs of $0.18 per kWh [26], the operating costs of a data center required for ChatGPT would have been $2.9\times$ higher in Europe than in the US.

$$564,000kWh \cdot 0.18 \frac{\$}{kWh} = \$101,520 \tag{1.2}$$

Due to the high energy costs in Europe, running this large AI data center would have entailed more significant costs than, e.g., the US. Thus, Europe has to overcome the energy costs challenge to keep up with the rest of the world's AI data centers [27]. Consequently, reducing the necessary energy consumption for AI tasks and improving sustainability should be one of the highest aims of European data centers. Worldwide, the data centers cause about 1 % of the energy use [25]. In 2021, the worldwide energy consumption was 25,343 terawatt hour (TWh) [28]. Consequently, 1 % of the worldwide energy consumption is 253.43 TWh, which is about 54.27 % of the total energy consumption of Germany with 467 TWh in 2023 [29]. As prognoses estimate a further increase in the portion of data centers' energy consumption worldwide, necessary actions like best practices or reuse of waste matter (e.g., heat) are required to improve sustainability. On the one hand, energy consumption can stay constant by applying best practices like the *4Ms* [23]. On the other hand, by reusing thermal dispatch, sustainable processes can be implemented in data centers.

The TDP of GPGPUs is primarily determined by their maximum power consumption. Therefore, ChatGPT requires a cooling system that can dissipate up to 564 MWh of heat per day. The heat generated by the data center can be utilized

for sustainable heating systems or wasted. New structural designs are necessary to achieve the former, such as connecting data centers to district heating grids to provide energy to nearby households. However, existing data centers must have the infrastructure to enable sustainable reuse.

An excellent example of the reuse of thermal discharge is the Eurotheum in Frankfurt, Germany. The 111-meter-high building is partly heated by the thermal discharge of the in-house data center operated by Cloud&Heat. The data center uses a water cooling system to bring around 600 MWh of thermal discharge into the heating system of the Eurotheum building. Therefore, the water is heated to 60 degrees Celsius (i.e., 140 degrees Fahrenheit). The 600 MWh thermal discharge would heat about 30 households for one year [30, 31]. Utilizing thermal discharge to partially heat the Eurotheum Building allows heating cost savings of up to 10 % annually [30, 31].

Many data centers currently use air cooling systems that follow a front-to-back principle. Cold air is drawn in from the front, and heated air is expelled from the back of the equipment. The air conditioning systems then capture the heated air and transfer it to a heat exchanger, which transfers the heat to water. However, this method only heats the water to temperatures ranging from 30 to 40 degrees Celsius, which is insufficient for heating systems. In such cases, additional thermal heat pumps are required to reuse the thermal discharge [30, 31]. Moreover, the water cooling system is more energy efficient than the air cooling system, especially during summer when outdoor temperatures rise. Air cooling systems require more energy to cool the infeed air temperature below 27 degrees Celsius, while water cooling needs to cool the water to 50 degrees Celsius to avoid heating hotspots in the machines. This makes water cooling more energy-efficient than air cooling, assuming a base level of 60 degrees Celsius for heat re-usage [31].

The potential of heat re-usage has yet to be exhausted. Structural changes and district heating systems are required to exhaust the potential. Therefore, communities and governments must create the necessary infrastructure to provide the heat of data centers for long distances; otherwise, only short-distance re-usage, like the Eurotheum, where the data center is in the same building, is possible. Overall, the following challenge in structural data center designs can be derived:

Challenge 2. ***Reduction of Energy consumption:*** *As energy consumption rises and becomes enormous, necessary improvements are necessary to remain sustainable. Consequently, the applications' energy consumption or the AI accelerator needs to be reduced.*

1.2 Challenges on Edge and IoT System Design for AI

In Edge and IoT systems, the most crucial challenge is to provide sufficient computational capacity while matching the specific limitations on thermal, energy consumption, or form factor size [32]. Because of the tight constraints and limitations of Edge and IoT systems, AI applications focus on the inference part. At the same time, the training steps are usually performed in the cloud or self-hosted data centers. Sustainable computing and energy consumption reduction are perfectly compatible with IoT and Edge device design as the limitations of Edge and IoT systems often include limited power supply (e.g., devices with battery). Consequently, sustainable computing can be incorporated into the design process and the overall requirements of IoT and Edge devices.

Although CPU and GPGPU are used for AI processing, GPGPUs cause higher power consumption. Thus, for applications requiring short computations, CPUs often provide enough computational capacity [7, 8, 32].

The computational requirements of Edge and IoT AI applications can be represented in two different types. First, the applications need to run constantly to meet the real-time constraint. For instance, in the case of video stream processing, this means processing the stream without frame drops. Second, applications that are latency sensitive [32]. Taking autonomous driving as an example, the latency of video processing should be less than 100ms for safe driving [32, 33].

Google spotted that about 60 % of the energy consumption related to AI applications from 2019 to 2021 was generated by inference tasks [23, 25]. The inference tasks are located from the cloud to IoT devices. However, Vries [25] spotted that the current research on power reduction is performed in the training process, and only a tiny part of the latest studies focus on the inference part of AI. Regarding Google's findings, most research considers the training process and

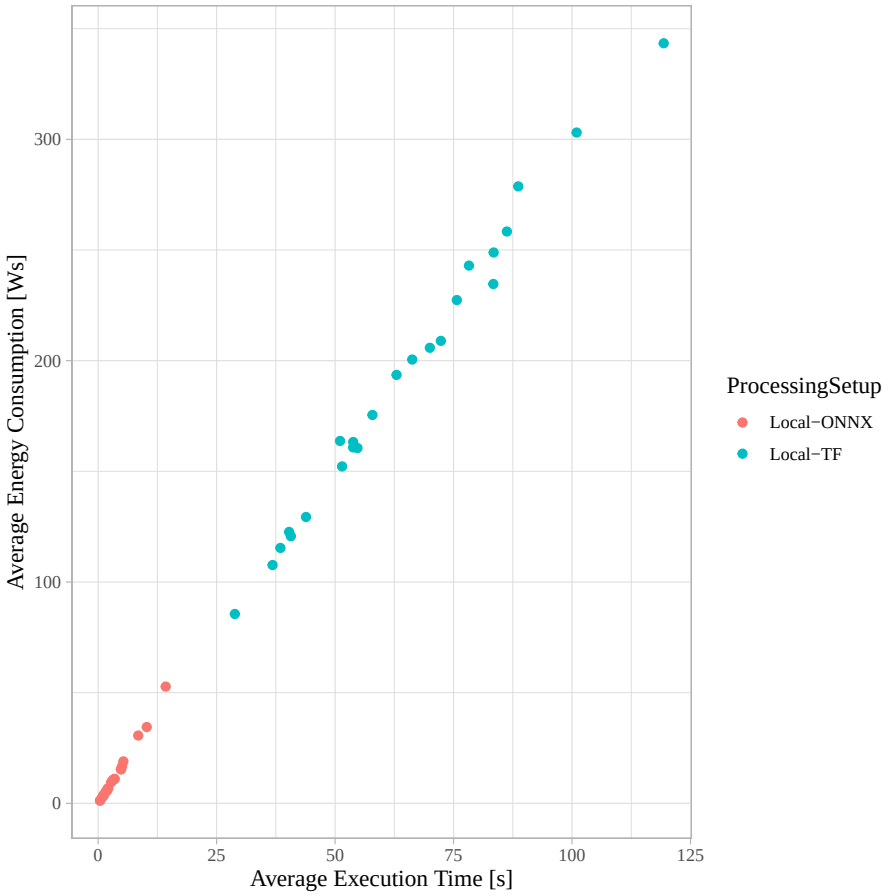


Figure 1.3: Comparing different neural network engine implementations on NVIDIA Jeston Nano 4GB.

focuses on the minor part of the cause of energy consumption. Consequently, more analysis on energy consumption reduction at inference must be performed.

Different implementations and frameworks can vary significantly regarding energy consumption and performance. Fig. 1.3 illustrates the average energy consumption and execution time of 32 different CNNs when performing inference with Tensorflow and Open Neural Network Exchange (ONNX). These CNNs have already been pre-trained, downloaded from TensorflowHub, and saved for TensorFlow execution. Furthermore, these same models have been exported to ONNX, and inferences have been made using the ONNX Python implementation.

Fig. 1.3 demonstrates that the average energy consumption of ONNX execu-

tion is significantly lower than that of Tensorflow. Moreover, it shows that ONNX execution is generally faster than Tensorflow execution. Therefore, it is essential to carefully consider the choice of framework when designing an AI system, as it can directly impact energy consumption and overall sustainability.

Overall, the following challenges are identified in sustainable computing for AI applications in Edge and IoT systems:

Challenge 3. ***Selection of AI accelerator for inference:*** *Most research focuses on the energy consumption of AI training. However, as the major energy consumption is generated by AI inference, more research on the inference is necessary. Latest studies [6, 7, 9] illustrate that the power consumption and performance of AI inferencing is predictable; thus, energy consumption can be determined. Consequently, based on the 4Ms practice, the selection process of AI-accelerators must be optimized for power consumption and performance of the AI application.*

Challenge 4. ***Energy efficient implementations:*** *As demonstrated, the energy consumption is significantly influenced by implementing the AI application. Thus, a crucial challenge is to optimize the implementation of AI inferencing task on energy efficient aspects. Different frameworks (e.g., ONNX or Tensorflow) and offloading strategies can substantially impact energy consumption. Hence, finding the best implementation regarding energy is of utmost importance to establish sustainable AI systems.*

1.3 State-of-the-Art Solutions

In the following, State-of-the-Art solutions are presented. This section is split into two parts: first, approaches for power and performance estimation are presented, and second, techniques for offloading computational-intensive applications or parts of applications to the cloud are presented.

1.3.1 Power and Performance Prediction

As the aforementioned challenges illustrate, the energy consumption of AI applications is a crucial aim to reduce. In the past, first solutions were presented to

estimate the power consumption and performance of GPGPU application in general and AI applications. The existing solutions to predict the power consumption of Compute Unified Device Architecture (CUDA)-based applications (i.e., an application developed for NVIDIA GPGPUs) can be divided into two main categories, which are: 1) the statistical analysis of the application and devices, e.g., [34, 35, 36], and 2) ML-based methods which use different ML algorithms to learn from a dataset and create a predictive model, e.g., [22, 37, 38, 39, 40]. Based on the literature [41, 42], ML-based methods provide better results in comparison to the statistical analysis and become the predominant technique to predict the power consumption of CUDA-based applications. Hence, this section gives an overview of ML-based methods and discusses their features and issues.

Existing ML-based methods use so-called performance counters as features to perform power consumption and performance estimation [22, 34, 37, 38, 39, 40]. Performance counters can only be collected and measured during run-time. This means an application must be executed once on a device to collect the performance counters. Afterward, the predictive model can run the prediction for other devices. This methodology can limit the usage in early design phases as a GPGPU must already be selected. Moreover, measuring performance counters requires a special GPGPU and CUDA profiler. Furthermore, performance counters are not unified across different devices and GPGPU models. Hence, a certain performance counter might not exist on a GPGPU or is measured differently [8].

The method in [37] uses tree-based regression to predict power consumption. It analyzes the GPGPU architecture and measures the power consumption of Parallel Thread Execution (PTX) instructions. However, as the method takes advantage of GPGPU-Sim, it is limited to a small subset of available PTX instructions. Moreover, the measured power consumption also relies on the predictive model of GPGPU-Sim as it is a simulation of GPGPUs executed on CPUs. Thus, the measured power consumption can differ from values measured on actual GPGPUs

The method in [43] considers scaling frequencies of GPGPU cores and memory for performance estimation. For power consumption estimation, it takes advantage of the method proposed in [40], which relies on *Support Vector Machines* (SVM) and performance counters. However, performance counters are

only available at run-time, limiting the use of this approach in early design steps. Moreover, the performance counters of NVIDIA GPGPUs are not unified across all models; consequently, the required performance counter might not be available on a certain model [8].

The method in [44] introduces an approach to estimate the performance (run-time) of CPU code before porting it to GPGPU code based on machine learning methods. This makes it possible to decide whether executing on GPGPU boosts performance. A similar goal is pursued in [45], but the prediction can already be performed on a GPGPU. The method uses CPU profile data and machine learning methods to estimate the run-time on GPGPUs. However, neither method supports power consumption prediction. They only consider run-time speed up between CPU and GPGPU.

PPT-GPU is a scalable GPGPU performance modeling system [46]. However, it does not yet support power consumption estimation. In [47], a layer-wise estimation approach is illustrated. It focuses on embedded GPGPUs and only considers platforms of the NVIDIA Jetson family. ALOHA [42] presents a statistical platform-aware evaluation method for CNN's execution on heterogeneous systems. For a given heterogeneous system and CNN, it can provide designers with operations and data transfers and their deployment on computing and communication resources. Moreover, it reports an estimation of latency and energy consumption of the CNN on the platform. However, the method requires an execution model that adequately describes the details of the platform and the scheduling of different CNN operators on different platform processing elements, which may not always be available. It is closely linked to hardware profiles, which are only available for Field Programmable Gate Array (FPGA)s and embedded GPGPUs.

1.3.2 Offloading

The following section will showcase the most advanced offloading approaches. When certain parts of an application, such as those related to AI, require significant computational power, they can be executed on remote machines. Even small devices like IoT can benefit from AI despite their limited computational resources.

DAvinCi [48] is a framework for robots to support the scalability and paral-

lelism advantages of cloud computing and implement the FastSLAM algorithm to provide Map-Reduce operations. Due to high latency and low energy efficiency between the devices and remote cloud, the advantages in real-world applications turn out low [49].

The Energy Efficient Computational Offloading Framework (EECOF) [50] focuses on computational offloading in Mobile Cloud Computing (MCC). The framework leverages application processing services of cloud systems with minimal instances of application migration at runtime. Energy consumption is reduced for mobile applications. EECOF is using *Inter Process Communication* (IPC) techniques such as Remote Procedure Call (RPC) or Remote Method Invocation (RMI). EECOF provides the computational parts as Software as a Service (SaaS) in a Service Oriented Architecture (SOA) to avoid time delays due to runtime deployment. Thus, there is no delay in deploying at runtime. Implementing the SaaS components before production requires additional development time.

In [51], an energy-efficient DNN placing approach is proposed for Unmanned Arial Vehicle (UAV). As UAVs are usually limited by resources such as storage, a placement problem occurs when distributing trained DNNs from the cloud to UAVs for inferencing. The proposed approach calculates the optimal placement to meet the resource requirements and provide the most energy-efficient solution.

1.3.3 Limitation of State-of-the-Art approaches

There are different ways to explore design space for ML inferencing computer architecture, but two main approaches stand out: 1) simulation and 2) ML-based predictors. However, both have their drawbacks. For example, simulators like GPGPU-Sim or GPU-ocelot run GPGPU applications on CPUs for simulation, which leads to significantly slower simulations than on real devices due to CPUs not having the same high parallelization ability as GPGPUs. ML-based predictors aim to provide fast and accurate estimations. Still, most require specific configuration and profiling of the application on a real GPGPU to collect necessary performance counters. Since performance counters are not standardized across all NVIDIA GPGPUs, it's possible that the required counter is unavailable or is

collected differently than in the original approach, making it impossible to apply the approach or lead to inaccurate results [8]. System designers require power and performance estimation approaches with straightforward collectible features to meet design constraints and improve sustainability.

As Challenge 3 *Selection of AI accelerator for inference* illustrates, it is necessary to investigate the power and performance behavior of AI inferencing tasks. Thus, the following research questions focus on designing AI systems for inferencing tasks. First of all, the system designer must decide whether applications should be offloaded (see Challenge 1 *the placement problem*). The State-of-the-Art approach does not offer a decision support system for various mobile network types that vary in bandwidth and latency. After system designers solve the placement problem and decide where to run the application, it is necessary to configure the device or platform for the AI application. The selection process for AI accelerators is especially sensitive as requirements for performance and power consumption can limit the valid options. This can be seen in Challenge 4 *Energy efficient implementations* and Challenge 3 *Selection of AI accelerator for inference*. Moreover, the selection of the AI accelerator also affects the total energy costs of a system. As Challenge 2 *Energy consumption* illustrates, it is of utmost interest to keep the power consumption low to reduce the energy consumption and carbon footprint, to improve sustainability as energy prices have risen in recent years. Hence, the following research question can be derived:

Research Question 1.1. *How is it possible to create rules to support decision-making on whether to execute AI applications on IoT and Edge systems or offload them?*

Research Question 1.2. *How can the power and performance for AI-based applications that are planned to be executed on GPGPUs be predicted with off-the-shelf techniques without the need for runtime-dependent features?*

Research Question 1.3. *What are the most significant impact factors on the power and performance of AI-based applications executed on GPGPU?*

Research Question 1.4. *Can machine learning-based power and performance estimation speed up the DSE for systems designed for AI-based applications on GPGPUs?*

1.4 Contributions

A series of techniques are created for designing IoT and Edge devices using computer-aided methods, surpassing the current limitations of State-of-the-Art approaches. The focus of this thesis is helping designers to select the most suitable GPGPU to improve the time-to-market span through analysis and ML-based predictive models. This thesis centers on designing inferencing systems for machine learning applications, emphasizing power and performance predictions and power-saving strategies. Therefore, the key contribution of this thesis can be categorized into five categories.

1. **Statistical Analysis and Decision Support System for AI offloading:** Challenge 1 illustrates, that it is important for IoT and Edge system designers to solve the placement problem. However, as the application, the network bandwidth, and the latency have a significant influence on the success of offloading, it is important only to offload if the determining factors match. This thesis presents a statistical analysis model and a decision support system for the technical setup. This model and decision support system enables the system designer to choose if offloading is a valid option and thus answer Research Question 1.1. Moreover, it can be implemented into IoT devices to make smart devices that can decide if offloading is an option during run-time. By delving into the details, the contribution is covered in Chapter 3.

2. **Power estimation model:** With the rising energy consumption (see Challenge 2) and considering IoT devices running on battery, it is mandatory to select an appropriate AI accelerator that matches the energy and power constraints. To avoid long prototyping cycles, this thesis presents a predictive model for power consumption prediction of CNNs based on ML. The predictive models achieve an Mean Absolute Percentage Error (MAPE) of 8.5 % and give necessary insights to answer the Research Questions 1.2, 1.3 and 1.4. Moreover, applying the predictive models avoids the need to run the AI application on real devices. The contribution is presented in Chapter 4.

3. **Power and performance analysis considering Dynamic Frequency Scaling (DFS):** Chapter 4 only considers power consumption of GPGPUs at their default configuration, it is necessary to analyze different GPGPU configuration settings in terms of power consumption and performance to answer the Research Question 1.3 precisely. Hence, Chapter 5 presents an in-depth analysis of the performance behavior of CNNs on GPGPU at different frequencies. The study illustrates that the frequency has barely to no influence on the performance (i.e., computation time) of different CNNs while having a significant impact on the power consumption. Considering Challenge 2 that the energy prices are rising, it is suggested to run CNNs with lower frequencies to reduce power consumption as the effect on the computation time can be neglected. This provides further insights to answer the Research Questions 1.2 and 1.3.

4. **Code analysis for data gathering:** To obtain data on the application's code, carefully examining the source code or running the application on actual hardware is necessary. As the predictive models in Chapter 4 and Chapter 5 rely on static code analysis, the data gathering has significant drawbacks, which can be overcome by profiling the AI application on real GPGPU or by the simulation of GPGPU. However, these solutions are time-intensive.

   To provide a fast and accurate alternative, static code analysis is integrated with simulation techniques to compensate the slow speed when simulating highly parallel code on hardware that does not support the same parallelism capabilities as GPGPUs. By doing so, the need for performance counters that may vary between different models of GPGPU is eliminated. As a result, it is possible to generate a set of easily collectible metrics giving important information to answer Research Question 1.4. The contribution can be found in Chapter 6.

5. **Performance estimation models:** The presented approaches provide various models to estimate performance of CNNs on GPGPUs. These models can also be applied to other types of Neural Network (NN). Using the provided models, the system designer can predict the performance of CNNs on

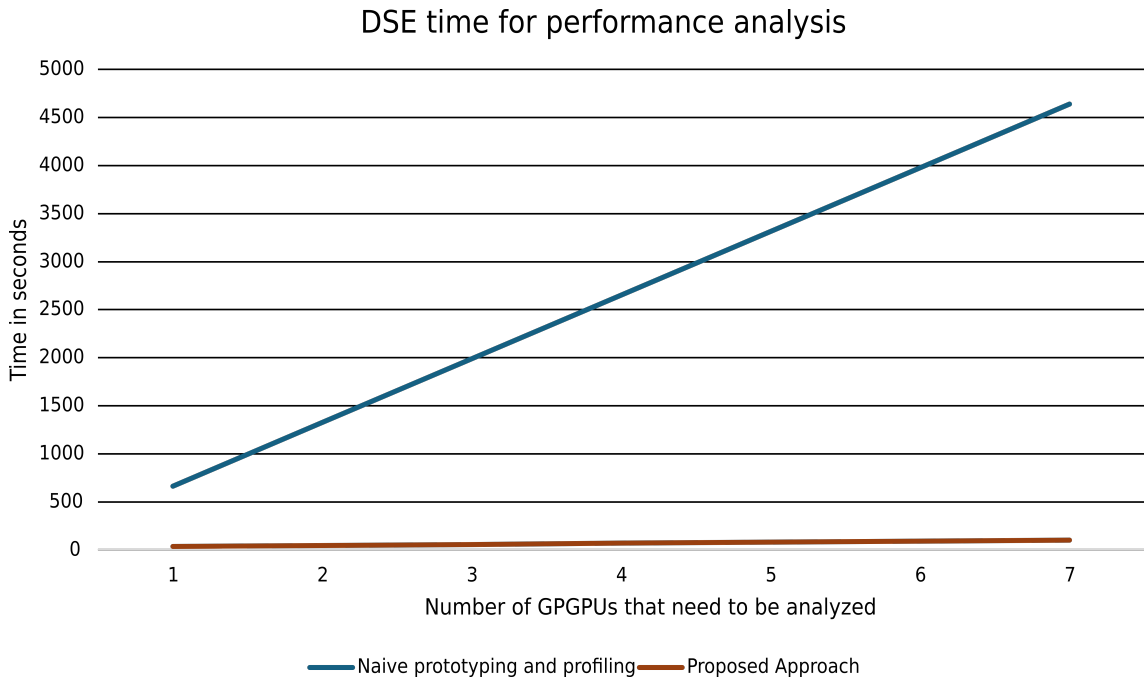


Figure 1.4: Improving the DSE analysis time by combining HyPA for data gathering and predictive models for performance estimation.

different GPGPUs. This allows a more efficient selection process that can save time and resources by reducing the need for multiple prototypes and improving the time-to-market span. As the contribution from Chapter 6 is incorporated into the predictive models, the performance can be estimated without any execution on real devices. Fig. 1.4 illustrates how much the DSE process can be improved regarding analysis time compared to State-of-the-Art approaches based on profiling on real devices. The naive prototyping approach requires multiple prototypes that need to be analyzed, and the application is profiled on the real device. Please note, that the time to build the prototypes is not included in the results illustrated in Fig. 1.4. The novel proposed approach uses predictive models based on ML to estimate the performance (i.e., number of cycles) and Hybrid PTX Analysis (HyPA) (detailed introduced in Chapter 6) for data gathering. This leads to an impressive increase in the number of GPGPUs that can be analyzed. The speedup is increasing with more considered devices, leading to a scaleable

approach and improving the time-to-market span. The prediction achieves a MAPE of 5.73 %, leading to an accurate performance estimation without any execution on a real device. The contribution is described in the following Chapter 7. Further insides to answer Research Question 1.2, 1.3 and 1.4 are provided in Chapter 7.

This thesis is based on the following peer-reviewed publications of the author:

1. [21] Christopher A. Metz, Mehran Goli, and Rolf Drechsler. Pick the Right Edge Device: Towards Power and Performance Estimation of CUDA-based CNNs on GPGPUs. *CoRR*, abs/2102.02645, 2021

2. [5] Christopher A. Metz, Mehran Goli, and Rolf Drechsler. Early Power Estimation of CUDA-Based CNNs on GPGPUs: Work-in-Progress. In *Proceedings of the 2021 International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '21, page 29–30, New York, NY, USA, 2021. Association for Computing Machinery

3. [6] Christopher A. Metz, Mehran Goli, and Rolf Drechsler. ML-based Power Estimation of Convolutional Neural Networks on GPGPUs. In *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 166–171, 2022

4. [7] Christopher A. Metz, Mehran Goli, and Rolf Drechsler. Towards Neural Hardware Search: Power Estimation of CNNs for GPGPUs with Dynamic Frequency Scaling. In *Proceedings of the 2022 ACM/IEEE Workshop on Machine Learning for CAD*, MLCAD '22, page 103–109, New York, NY, USA, 2022. Association for Computing Machinery

5. [9] Christopher A. Metz, Mehran Goli, and Rolf Drechsler. Fast and Accurate: Machine Learning Techniques for Performance Estimation of CNNs for GPGPUs. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 754–760, 2023

6. [8] Christopher A. Metz, Christina Plump, Bernhard J. Berger, and Rolf Drechsler. HyBrid PTX Analysis for GPU accelerated CNNs inferencing

aiding Computer Architecture Design. In *2023 Forum on Specification & Design Languages (FDL)*, 2023; **Honored with the Best Paper Award**

7. [52] Rolf Drechsler and Christopher A. Metz and Christina Plump. Energy-Efficient CNN inferencing on GPUs with Dynamic Frequency Scaling. In *Innovations in Data Analytics*, 2023

The present thesis is based on works that are currently undergoing review. These works have exerted a direct influence on the development of the thesis. However, it is important to note that final publication versions of these works may vary from those utilized in the present thesis:

1. [53] Christopher A. Metz and Rolf Drechsler. Challenges on Sustainable Artificial Intelligence Inference. *Communication of ACM*, 2024

2. [54] Christopher A. Metz, Christina Plump, Bernhard J. Berger, and Rolf Drechsler. Performance and Energy Tradeoff Analysis of CNN Offloading to the Edge and Cloud Computing. In *2024 IEEE International Conference on Edge Computing and Communications (EDGE)*, 2024

Moreover, the following articles are not included in this thesis but influenced its preparation.

1. [20] Christopher A. Metz and Ralf Buschermoehle. Aufbau eines Machine Learning Clusters fuer Forschung und Lehre. In *20. DINI Jahrestagung*, 2019

2. [55] Sana Hassan Iman, Christopher A. Metz, Lars Hornuf, and Rolf Drechsler. Classifying Crowdsourcing Platform Users' Engagement Behavior using Machine Learning and XAI. *In Mensch und Computer 2023 - Workshopband*, 2023

3. [56] Christopher A. Metz. Machine Learning aided Computer Architecture Design for CNN Inferencing Systems. *arXiv preprint arXiv:2308.05364*, 2023

4. [57] Sana Hassan Imam, Christopher A. Metz, and Rolf Drechsler. How Can Generative AI Curate the User Creativity on an Idea Crowdsourcing Platform? In *ACM CHI 24 Workshop on Generative AI in User-Generated Content*, 2024

2 Background

As the thesis focuses on inferencing of CNN on GPGPUs and offloading strategies for CNNs in IoT and Edge environments, the necessary preliminaries are presented in the following chapter. Energy consumption metrics are presented in Section 2.1. Afterward, the concept of NN, ML exchange formats and GPGPUs are explained in Section 2.2, Section 2.3 and Section 2.4, respectively.

2.1 Energy Consumption Metrics

Energy is the total power consumed during an interval of time and measured in joules (J) or watts per hour (Wh). The ratio is $1Wh = 3,600J$. Consequently, the energy consumption can be calculated as follows for general cases:

$$E = U \cdot I \cdot t \tag{2.1}$$

where $U, I, andt$ denotes the voltage in Volts (V), the current in Ampere (A), and the time. Thus, the power consumption in Watt is calculated as:

$$P = U \cdot I \tag{2.2}$$

Power consumption rate can be split into *static* and *dynamic power* consumption. *Static power* is the power consumed during the idle process of the circuit and is also known as leakage power. *Dynamic power*, on the other hand, is the power consumption caused by the activity of the circuit [58]. The dynamic power consumption for a circuit can be calculated as follows:

$$P_{dynamic} = ff \cdot C \cdot V^2 \cdot f \tag{2.3}$$

where ff is the activity factor in percentage. V is the voltage, C the capacitance, and f the clock frequency [58]. The total power consumption is the sum of the static and dynamic power consumption. Each hardware component has its own static power consumption. However, a system's exact energy and power consumption is based on the application, the involved hardware components, and how these components are used. Thus, defining the equation to calculate the energy consumption is difficult [7, 58].

Besides this metric, recent studies used so-called performance counters, a set of special-purpose registers in modern processors, to specify energy efficiency. Performance counters are counting specific event types that are hardware related (e.g., L2 cache misses) [58]. However, performance counters are manufacturer-specific and cannot be used for manufacturer overlapping analysis. Moreover, in the case of NVIDIA GPUs, the performance counters also differ between the different NVIDIA GPUs, making it even more difficult to compare different components in terms of energy efficiency based on performance counters [8].

2.2 Neural Networks

Nowadays, AI is a thrilling topic leading to many practical applications and creating active research topics [10]. A significant field of AI is Deep Learning (DL) which solves central problems in representation learning [10]. The quintessential for DL are multilayer perceptron. A multilayer perceptron is a mathematical function mapping a set of inputs to one output value [10]. A single perceptron is also referred to as a neuron. With the concept of multilayer perceptrons, more complex types of networks can be built; these are often referred to as NN or DNN. Moreover, different network types and topologies have been developed in the last decades.

The origin idea of a perceptron was developed in the 1940s-1960s by McCulloch and Pitts [10, 59]. A NN is built by several neurons (perceptrons) composed in a layer. A NN can consist of multiple layers but has at least two: an input layer and an output layer [10].

The success of NNs can be traced back to the large data set generated by Big

Data and IoT technologies and increasing computing capabilities due to modern accelerator and processing units like GPGPUs. With the new accelerators, it is possible to run calculations for larger NNs (e.i., more layers of neurons) [10]. Over the years, different types of NNs have been developed to perform specific tasks. Some of them, like Rekurrent Neural Network (RNN), are ideal for processing time series data and Natural Language Processing (NLP) tasks. Others, like CNNs, are highly effective in image and video processing. However, both types of NNs require a significant amount of computing resources. Therefore, they are excellent candidates for benchmarking and demonstrating power-saving options compared to performance optimization techniques. For this thesis, CNNs will be used as a reference type in the following chapter to showcase the developed methodologies.

2.2.1 Convolutional Neural Networks

CNNs are mainly designed for image classification or recognition but are used in many more areas. They differ in size, accuracy, and complexity depending on the use case, the number of layers, and neurons per layer [60, 61]. The so-called trainable parameters (also called weights) are one option to describe the complexity of a CNN. They are the number of connections of a NN [62]. Fig. 2.1 illustrates the general architecture of a CNN model with n input and m output neurons.

In the case of a convolutional layer, the trainable parameters depend on the number and size of the kernel used in the convolutional layer. A convolutional layer can have three stages, which are 1) convolution, 2) activation, and 3) pooling. In the first stage, several convolutions are executed in parallel. The second stage applies the linear activation function, such as Rectified Linear Unit (ReLU) activation function. The third stage performs a pooling function such as the max-pooling [10]. This stage is optional and not included in all convolutional layers. For example, the Alexnet has three max-pooling layers and six convolution layers [63]. The max-pooling function selects the highest value from a kernel or window with $n \times m$ size. Technically, the second and third stages do not have trainable parameters. However, to calculate the trainable parameters for a fully connected layer

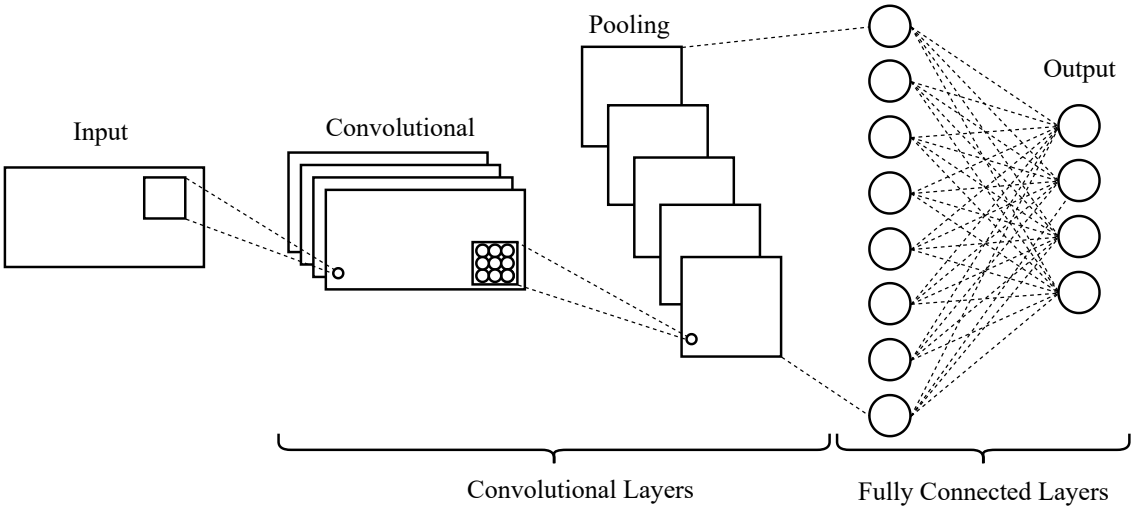


Figure 2.1: The general architecture of a CNN model.

following a convolutional or pooling layer, it is necessary to calculate the output dimension of the convolutional or pooling layer first. As a consequence, the architectural parameters of the pooling layer are important, too. In fully connected layers, each layer node has n connections, one to each node of the following layer. As a consequence, the fully connected layers at the end of the CNN in Fig. 2.1 have $n \times m$ weighted connections. To obtain the total number of trainable parameters of the CNN, the connections between each model layer must be considered.

The following describes how the trainable parameters for a CNN can be calculated [10]. Please note that most DL frameworks already include functionality to calculate the trainable parameter. Hence, in most cases, it is not necessary to perform the calculation from scratch.

2.2.2 Benchmark CNNs

As the CNNs are used as an example class of NNs throughout this thesis, this section introduces the CNNs, which are used for the Benchmarks and experimental evaluation in the following chapters. Table 2.1 gives an overview of the 32 different CNNs and their structure. As Tab. 2.1 illustrates, most CNNs have an input layer size of 224×224; this is because most of the CNNs are trained on

the same training dataset; namely ImageNet [64]. However, to create a versatile benchmark, CNNs with different input layer sizes are considered.

Eleven CNNs are out of the ResNet family and based on the *Deep Residual Learning for Image Recognition* architecture [65]. As these CNNs distinguish their large number. Two networks were created using Neural Architecture Search (NAS) technology. The NASNetmobile [66] is designed to run on mobile devices. The Efficientnetb0 to 6 [67] are characterized by varying input data sizes and the number of hidden layers and neurons.

Overall, the selected meshes are diverse in their characteristics and, therefore, provide a good basis for benchmarking GPGPUs to create a training data set. To ensure that all results are reproducible, pre-trained implementations of the networks were downloaded from TensorFlow Hub.

2.3 Machine Learning Exchange Formats

Model Request API (MoReA) is working on making it easier for developers to access different machine-learning models. In the following, two formats that aim to increase machine learning models' portability are introduced, enabling MoReA to support different machine learning models.

2.3.1 Open Neural Network Exchange

ML frameworks, such as Torch, Caffe, Theano, or TensorFlow, support developers in model training and inference. However, it is usual to do these tasks on different systems. For example, a developer trains a model on an extensive HPC system but performs the inference task on a small IoT device due to the system's requirements. The different execution environments of these tasks may result in the problem that the programming language or framework used during training might not be available on the inference device. The ONNX solves this problem by defining a format to transfer ML models between different systems more simply [68]. Therefore, the target platform must provide an ONNX runtime for loading and executing the stored model.

Table 2.1: An overview of CNN models used in the experiments

Model name	Input Size	Layers	Neurons	Weights
m-r50x1	224×224	50	15,903,016	25,549,352
m-r50x3	224×224	50	143,111,080	217,319,080
m-r101x3	224×224	101	25,3408,168	387,934,888
m-r101x1	224×224	101	28,158,248	44,541,480
m-r154x4	224×224	154	611,981,544	936,533,224
resnet101	224×224	101	55,886,036	44,601,832
resnet152	224×224	152	79,067,348	60,268,520
resnet50v2	224×224	50	31,381,204	25,568,360
resnet101v2	224×224	101	51,261,140	44,577,896
resnet152v2	224×224	152	75,755,220	60,236,904
nasnetmobile	224×224	771	27,690,705	5,289,978
nasnetlarge	331×331	1041	290,560,171	88,753,150
densenet121	224×224	121	49,926,612	7,978,856
densenet169	224×224	169	60,094,164	14,149,480
densenet201	224×224	201	77,292,244	20,013,928
mobilenet	224×224	28	16,848,248	4,231,976
inceptionv3	299×299	48	32,554,387	23,817,352
vgg16	224×224	16	15,262,696	138,357,544
vgg19	224×224	19	16,567,272	143,667,240
efficientnetb0	224×224	240	25,117,095	5,288,548
efficientnetb1	240×240	342	40,150,331	7,794,184
efficientnetb2	260×260	342	50,908,981	9,109,994
efficientnetb3	300×300	387	87,507,971	12,233,232
efficientnetb4	380×380	477	180,088,531	19,341,616
efficientnetb5	456×456	579	358,290,427	30,389,784
efficientnetb6	528×528	669	605,671,091	43,040,704
efficientnetb7	600×600	816	1,046,113,195	66,347,960
Xception	299×299	71	62,981,867	22,855,952
MobileNetV2	224×224	53	21,815,960	3,504,872
InceptionResNetV2	299×299	164	81,201,907	55,813,192
alexnet	227×227	8	650,000	58,325,066
resnet50	224×224	50	25,612,201	210,767,874

2.3.2 Save and Load TensorFlow Models

TensorFlow offers different approaches to save and load models, ranging from checkpoint to restart the training process from those save stages to exchange able model formats[1]. TensorFlow offers a high-level format and an advanced format included in the Keras Application Programming Interface (API). However, as the TensorFlow model is code, it is important to be careful with untrusted sources. A reliable source of TensorFlow models is TensorFlow Hub[2]. The models used in this study rely on the high-level Keras format for TensorFlow models.

2.4 Graphic Processing Units

GPGPUs have different and more complex architecture compared to traditional CPUs. The GPGPU architecture consists of a scalable number of Streaming Multiprocessors (SMs). Fig. 2.2 illustrates the SM of the NVIDIA V100. To improve the utilization, the SM is partitioned into multiple Processing Unit (PU)s. While the SM is divided into two PUs in the predecessor (NVIDIA P100), this number increased to four for the NVIDIA V100. Each PU containing 16 Floating Point (FP) 32 Cores, 8 FP64 Cores, 16 INT32 Cores, one Tensor Core (with mixed-precision Tensor Cores for Deep Learning), a L0 instruction Cache, one warp scheduler, one dispatch unit, and a 64KB Register File [35, 69].

Kernels compiled for a GPGPU are subdivided into Cooporate Thread Arrays (CTAs), also called thread blocks [70]. A CTA is further divided into groups of 32 threads called a warp [71]. All threads inside a warp are executing the same instruction [72]. The principle is similar to Single Instruction Multiple Data (SIMD). However, NVIDIA calls it Single Instruction Multiple Thread (SIMT). Unlike SIMD instructions, the concept of warps is not exposed to the programmers. Instead, programmers write a program for one thread and then specify the number of parallel threads in a block and the number of blocks in a kernel grid [35]. Handling the concept of warps as a black-box make code optimizing difficult.

The NVIDIA Tesla, Volta, and Ampere architecture form a warp using a batch of

[1] `https://www.tensorflow.org/tutorials/keras/save_and_load`
[2] `https://www.tensorflow.org/hub`

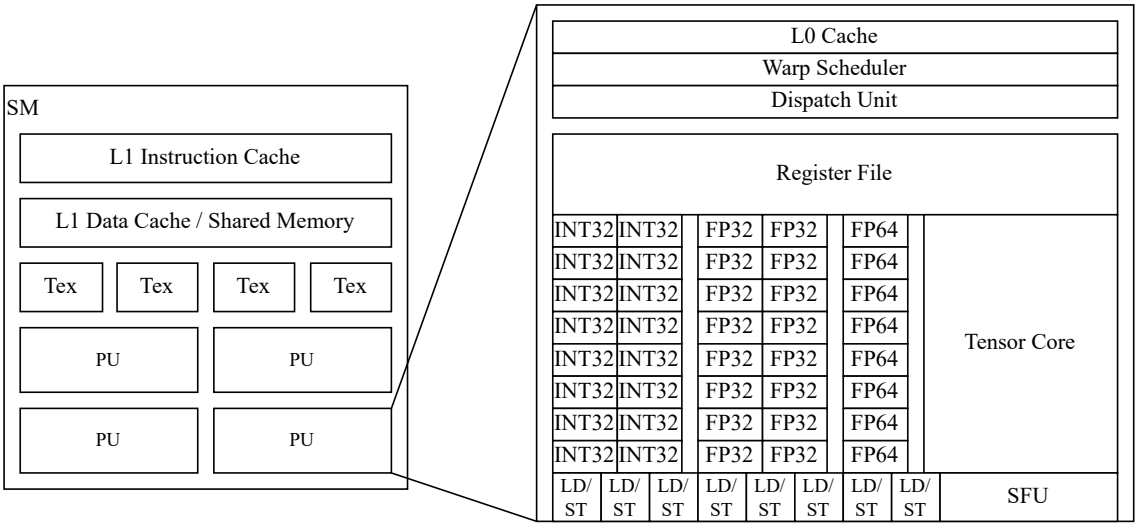


Figure 2.2: Overview of Streaming Multiprocessor architecture

32 threads [35, 73, 74]. All threads in one wrap are executed on one SM together. An SM can also handle multiple wraps [70, 71]. The number of concurrently running wraps is determined by the resource requirements of each wrap, such as the number of registers or shared memory usage. Since the Volta architecture, the warp-synchronous programming, that threads executing as part of the same warp are implicitly synchronized at every instruction, has been obsoleted [70].

NVIDIA provides a detailed natural language description of the PTX model. The analysis in this thesis aims to place NVIDIA GPGPU architectures starting from Volta on a solid and more reliable theoretical foundation since there are massive changes between the Volta architecture and their predecessor [70]. To test the portability to older GPGPU generations, we consider a GTX 1080Ti based on the pascal architecture.

Parallel Thread Execution

NVIDIA offers the CUDA Library to run applications on their GPGPUs. However, frameworks like Tensorflow already include the CUDA Library, so users can quickly run their applications on GPGPUs [75]. The code is compiled to the PTX to execute the CUDA applications on GPGPU. PTX is an Instruction Set Archi-

```
1  .visible .entry example(.param .u32 param_0)
2  .reqntid 256, 1, 1
3  {
4    .reg .pred    %p<3>;
5    .reg .u32     %ru<4>;
6
7    mov.u32       %ru1, %tid.x;
8    ld.param.u32  %ru4, param_0;
9    setp.lt.u32   %p1, %ru4, 1024;
10   setp.lt.u32   %p2, %ru1 128;
11   @%p1 bra      then;
12   mov.u32       %ru3, 32;
13   bra.uni       exit1;
14
15 then:
16   mov.u32       %ru3, 64;
17
18 final:
19   setp.gt.u32   %p3, %ru3 100;
20   @%p2 bra      exit1;
21   add.u32       %ru4, %ru4, 1024;
22
23 exit1:
24   ret;
25 }
```

Figure 2.3: Example PTX file.

tecture (ISA) including every memory access (read and write) and computational instructions (e.g., ADD, MUL, FMA) and is translated nearly one-to-one with native binary micro-instructions [76]. Some PTX instructions cannot be translated to one binary micro-instruction and are built of multiple [35]. In Fig. 2.3, a short extraction of a PTX assembly code is illustrated. The PTX ISA is translated to SASS, the target machine language of the GPGPU [77]. PTX is designed as virtual ISA to be portable between different GPGPU generations with different instruction sets (e.i., SASS implementations). The lack of portability makes SASS unattractive for analysis [78] as it limits the analyses to one GPGPU.

Divergent Branches

The PTX ISA and NVIDIA's GPGPU architectures allow so-called Divergent Branches. A Divergent Branch is a branch where some threads are within the same warp branch while others are not. Against the concept of SIMT, not all threads execute the same number of instructions [72, 79, 80]. This leads to an unknown amount of dependency instructions that cannot be considered by static code anal-

ysis.

Two cases of Divergent Branches can occur: 1) IF without ELSE; during this case, some threads enter the IF and execute the additional instructions, while others are idle due to the SIMT concept, where all threads have to execute the exact instructions. 2) IF, with ELSE, this case is more complex; while some threads enter the IF and execute the instructions, others are idle. Afterward, when the previous idle threads enter the ELSE statement, the IF-threads will become idle [72]. Both cases can lead to a different number of executed instructions during the single threads; hence, counting the instructions of a PTX code and multiplying them by the number of threads cannot lead to the exact number of instructions. In this work, we focus on identifying divergent branches and calculating the number of executed instructions, considering all dependency instructions within the Divergent Branches.

3 Performance and Energy Trade-off Analysis of CNN Offloading

Nowadays, ML achieves incredible results in almost all areas of everyday life, like object recognition or natural language processing. As a result, more and more ML applications enter products, especially in the industrial area of the IoT. However, ML techniques like NN are computationally expensive and often require dedicated ML accelerators. Consequently, the power consumption of IoT devices is increasing due to such accelerators. When power supply is a limiting factor, applications offload ML tasks to the cloud or edge to meet the trade-off between speed and power consumption.

This chapter presents the results of [54], an analysis of the trade-off between performance and energy consumption when offloading CNNs. It compares local implementations against offloading on five different mobile network types. The evaluation shows a significant influence of the network type and CNN complexity on the offloading-induced energy consumption improvement. After careful analysis, a decision support scheme is provided to choose between local and remote execution.

The remainder of the chapter is structured as follows: Section 3.1 starts with a brief introduction followed by Section 2.3 presents different Machine Learning Exchange Formats to discuss the necessary capabilities of the proposed API. Section 3.2 describes the introduced API and its features. In Section 3.2.2, the modeling of the offloading process is presented, and the influencing parameters, as well as investigated metrics, are discussed. Section 3.3 explains the experimental setup and describes the results obtained by these experiments. The pre-

sented interpretation of the described results and a detailed discussion of threats to the validity of the introduced modeling approach and the experimental setup and results. Finally, the chapter concludes with an outlook in Section 3.5.

3.1 Introduction

In recent years, ML methods have been used increasingly [81, 82]. For example, object recognition or image classification are upcoming applications in industrial IoT [24, 83]. As a result, IoT devices have adapted to the demand and are increasingly using accelerators optimized for ML [82]. Examples include the NVIDIA Jetson product line, a System-on-a-Module (SoM) that can run ML applications energy-efficiently using an NVIDIA Tegra GPGPU [82, 84]. However, IoT devices often have limited battery life but generate massive amounts of data to process [85]. Thus, energy-efficient applications with a more powerful accelerator are crucial.

Motivation New challenges arise with using ML models in IoT devices [86]. The needed computational resources for different ML approaches are broad [81]. While simple decision trees require few computational operations, CNNs require many [8]. This excessive computational effort results in performance losses when using small IoT devices with classical processing units. Considering that much more powerful ML-accelerators (e.g., GPGPUs) will increase the power consumption of the IoT device, as ML accelerators require high power consumption, their usage in IoT area is only practicable in cases with sufficient power supply [5, 6, 49]. Hence, offloading into Edge or Cloud computing systems is a promising strategy for these challenges [50, 87]. Studies illustrate that offloading computational-intensive applications to cloud or edge computing can significantly reduce the power consumption of IoT devices. Therefore, the computational-intensive parts are executed on cloud or edge servers to move the battery life-draining computation to remote systems [86, 88, 89]. However, this is only possible if the time delay is acceptable [88]. Thus, the power-time trade-off is fundamental when offloading applications or computations to the cloud or edge.

Limitation of state-of-the-art Most proposed approaches focus on finding the

power-delay trade-off while offloading ML models to edge computing [90]. Expanding on their findings, this chapter presents an analysis of offloading using a framework that provides models in a zero-programming way. In previous studies [6, 48, 50, 90], the availability of different types of mobile networks was not considered. The bandwidth and latency of the network available influence the potential of offloading. Therefore, it is essential to evaluate the behavior of different CNNs on various network types since a detailed evaluation can provide valuable insights for implementing offloading techniques into IoT devices.

Key insights and contributions Our key contributions presented in this chapter are (1) a thorough modeling and statistical analysis of the offloading scenario for CNN-based inferencing tasks, (2) the derivation of a guideline (based on the introduced setup) for deciding whether to offload or not (3) a flexible RESTful API allowing to offload ML tasks to the cloud using an innovative model description language. It was found that fast networks almost always yield a decrease in energy consumption when offloading; however, for simple CNNs, slow networks render this advantage mute. Computation time, on the other hand, mainly increases; however, for fast networks, the additional computation time is countered by the faster computation in the cloud, making offloading the efficient choice for both metrics.

Experimental methodology and artifact availability For the trade-off analysis, the different connectivities, inferencing tasks, and CNN parameters are modeled. Therefore, the energy consumption and computation time for the local and remote settings are measured, and statistically, the influence of several parameters on the measured metrics is analyzed. To enable a smooth offloading process, MoReA is used, which is developed for this purpose. MoReA will be open source, enabling individual customization and further development. Additionally, the evaluation data and scripts will be available to allow reproducibility[1].

Limitation of the proposed approach The main limitation of our evaluation and final decision guideline is the missing variation of local and remote machine performance. Additionally, in this study, it was found that at least one key parameter must be missing as it can not explain the behavior of the energy consumption of one given setup. The proposed guideline has been revised accordingly to ensure

[1] Upon acceptance

its validity is not affected; however, the generalisability of the findings is affected as a reliable prognosis cannot be made for this specific exchange format.

3.2 Methodology

The MoReA is a RESTful API and thus offers several Hyper Text Transport Protocol (HTTP) endpoints to enable clients to query provided ML models. MoReA offers a unique Model Description Schema (MDS), allowing the users to describe available ML models for MoReA. Thus allowing MoReA to support various ML frameworks and techniques, e.g., Decision Trees, NNs, and K-Nearest Neighbor (K-NN). The methodology is structured into two phases, as seen in Figure 3.1. First, model preparation and uploading. During this phase, the user writes a Model Description File (MDF) based on the MDS. The user then uploads the model and its corresponding description file to MoReA. Second, loading and serving. In this phase, MoReA automatically loads all available models and reads the belonging MDFs. An HTTP endpoint for each model is generated based on the details included in the MDFs. Consequently, any HTTP-enabled device can use the ML models, which allows MoReA to be used on IoT, Mobile, and many more applications.

3.2.1 Model Description Schema

Each ML model has a unique set of required input values with different data types, which are—in machine learning—often referred to as input features. Since only some ML frameworks offer the functionality to read the required input features and produced output values of an ML model, a description is inevitable to understand how to use a model for the inferencing process. Moreover, having a detailed description of the ML model MoReA can provide clients with a machine-readable model description. This description enables clients to read necessary details and generate input data for any model without explicit programming. Thus, the usage of MoReA is flexible and customizable for the users' needs.

This chapter presents the MDS capable of describing any ML model. It in-

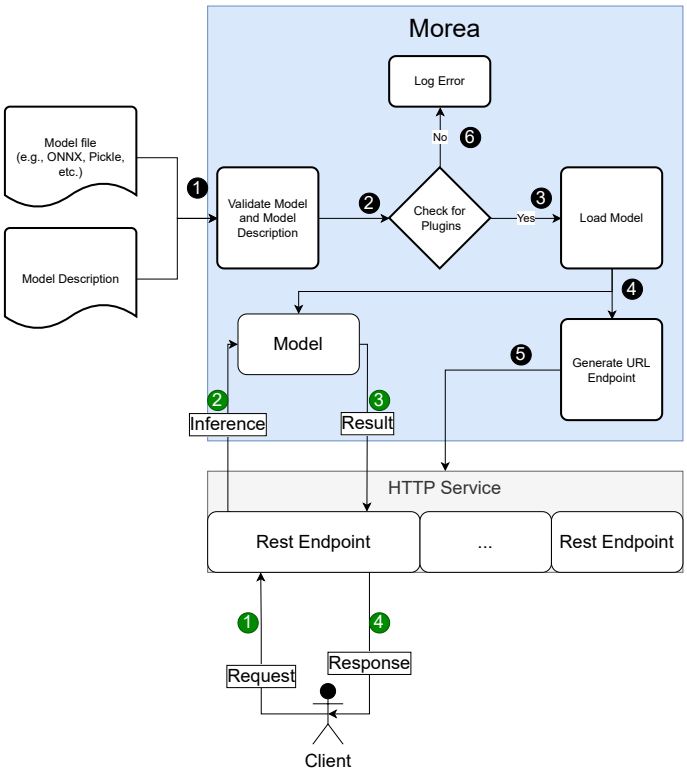


Figure 3.1: Overview of the functionality of MoReA core and HTTP service.

cludes basic information about the model's purpose, type, and required input features. Figure 3.2 shows the general structure of the MDF. As Figure 3.2 illustrates, for each model, the *modelName* has to be specified as it is reused for the Representational State Transfer (REST)/HTTP endpoint generation. Consequently, the *modelName* must be unique. Moreover, the *modelType* must be specified (e.g., ONNX, Tensorflow) to consider the correct execution engine while loading the models.

When MoReA loads all models in the second phase, it verifies the MDF's conformance to the MDS. The validation process includes determining if all required attributes are present. Afterward, each input feature is verified individually. Each input feature must have a name and kind attribute. There are two ways of describing the data for model input or output: first, by describing each data value in a single object. Second, the description can proceed nested in cases of multi-dimensional data, like images. The shape requires a description, size, and con-

```
{
    "modelName": "Name of the Model",
    "detail": "Detailed description of the model",
    "outputs": {
      "outputs": [...]
    },
    "modelType": ...,
    "inputFeatures": {
      "features": [...]
    }
  }
```

Figure 3.2: General model description structure for ML-model description

tent. The content can either be a shape description or the final data type of the multi-dimensional data. If the model does not fulfill one of the conditions, MoReA will not load it. Furthermore, MoReA skips the load process of a model if the model's name is not unique and another model with the same name already exists.

3.2.2 Statistical Modeling

This section presents the understanding of the offloading process and discusses the parameters that affect the energy consumption of the local machine and overall computation time in the inferencing task. In general, the following parameters are considered: `machine` (local, remote), `task`, `cnn`, `format`, and `connectivity`. `Machine` refers to the general performance capabilities of the local machine and the remote machine. `Task` refers to the complexity of the inferencing task, e.g., the size of the image sent to the CNN in an image recognition task. It is assumed that the `CNN` also influences the energy consumption and computation time. Therefore, this is an influencing variable. Strongly connected is the form of representation of the inferencing model, i.e., which exchange format is used, denoted with `format`. As the study deals with the question of whether or not to offload an inferencing task, it also considers connectivity (i.e., latency and bandwidth) as an influencing variable to the decision.

To enable fair comparisons and enhance the understanding of results, the steps necessary for performing the inferencing task are grouped into four distinct phases: SETUP, PROCESSING, POSTPROCESSING, END. The SETUP phase

contains all the necessary steps on the local machine to prepare for the inferencing task. The PROCESSING phase contains the actual inferencing, as it affects the local machine, i.e., it also contains the transfer time from the local to the remote machine. The POSTPROCESSING phase collects all (possibly) necessary steps to make the inferencing task result ready for the user, while the END phase closes connections and cleans up for the following tasks.

Local Inferencing

The inferencing task on the local device requires the following substeps:

1. Load required model (SETUP)
2. Prepare inferencing task for inferencing (SETUP)
3. Perform inferencing (PROCESSING)
4. Post-process result for end-user (POSTPROCESSING)
5. End inferencing (END)

The following influence of the considered parameters on the computation time t and energy consumption p are expected, which are tried to validate in the evaluation:

$$
\begin{aligned}
p_{\text{SETUP}}, t_{\text{SETUP}} &\leftarrow \mathrm{m}_{local}, \texttt{task}, \texttt{cnn}, \texttt{f} \\
p_{\text{PROCESSING}}, t_{\text{PROCESSING}} &\leftarrow \mathrm{m}_{local}, \texttt{task}, \texttt{cnn}, \texttt{f} \\
p_{\text{POSTPROCESSING}}, t_{\text{POSTPROCESSING}} &\leftarrow \mathrm{m}_{local}, \texttt{task}, \texttt{f} \\
p_{\text{END}}, t_{\text{END}} &\leftarrow \mathrm{m}_{local}, \texttt{cnn}, \texttt{f}
\end{aligned}
$$

Shortly, the reasoning for the above dependencies will be explained. It is easy to see that neither the remote machine nor the connectivity influences the local performance. For setup, however, the model needs to be loaded, so it is assumed an influence of the CNN model. At the same time, the inferencing task needs to be prepared, which leads to the influence of the inferencing task for the SETUP phase. Both the task and the CNN model influence the PROCESSING, together

with the properties of the local machine. In POSTPROCESSING, however, the model is not relevant anymore as this only takes care of the result. The final phase END, on the other hand, frees the model, so there might be some influence of the CNN here. As all tasks are performed on the same machine (local), the dependencies are identical for energy consumption and computation time.

Remote Inferencing

The inferencing task using a remote server requires the following substeps:

1. Prepare inferencing task for Inferencing (SETUP)
2. Send inferencing task to remote server (PROCESSING)
3. Load required model on remote server (PROCESSING)
4. Perform inferencing on remote server (PROCESSING)
5. Send result to local device (PROCESSING)
6. Post-process result for end-user (POSTPROCESSING)
7. End inferencing (END)

For this case, the following influence of parameters on the computation time t and energy consumption p are expected, where `m` is short for `machine`, `f` for format and `c` is short for `connectivity`:

$$
\begin{aligned}
p_{\text{SETUP}}, t_{\text{SETUP}} &\leftarrow \mathtt{m}_{local}, \mathtt{task}, \mathtt{f} \\
t_{\text{PROCESSING}} &\leftarrow \mathtt{m}_{remote}, \mathtt{task}, \mathtt{cnn}, \mathtt{c}, \mathtt{f} \\
p_{\text{PROCESSING}} &\leftarrow \mathtt{m}_{remote}, \mathtt{m}_{local}, \mathtt{task}, \mathtt{cnn}, \mathtt{c}, \mathtt{f} \\
p_{\text{POSTPROCESSING}}, t_{\text{POSTPROCESSING}} &\leftarrow \mathtt{m}_{local}, \mathtt{task}, \mathtt{f} \\
p_{\text{END}}, t_{\text{END}} &\leftarrow \mathtt{m}_{local}
\end{aligned}
$$

The reasoning behind this analysis is as follows: In the offloading setup, the model does not have to be loaded on the local machine, and this step is, therefore, not part of the SETUP phase. Hence, energy consumption as well as computation time are not dependent on the `cnn` anymore. The PROCESSING phase

is somewhat more complicated: The computation time depends on the `task` and the `cnn`, obviously, as well as the remote `machine` and the `connectivity`. It does not depend on the local `machine`. The energy consumption, however, directly depends on the local `machine` and the computation time and therefore inherits all dependencies from computation time. The POSTPROCESSING phase is equivalent to the local setup, while the END phase only depends on the local machine as the model has been taken care of on the remote site.

The overall goal of the presented research is to provide the local system with decision support for the question: Should this task be offloaded or performed locally? This leads to the following Research Goal of this chapter:

Research Goal 1. *Derive a statistically sound decision support for inferencing locally or remotely, given knowledge about influencing factors.*

To achieve this research goal, the influence of the above-mentioned factors on computation time and energy consumption must be analyzed. Additionally, the results must be generalized to derive statistically sound decision support.

Therefore, the following research questions are considered for this chapter:

Research Question 3.1. *How does the `connectivity` influence the energy consumption and computation time in the remote setup?*

Research Question 3.2. *How does the `cnn` influence the energy consumption and computation time in the local or the remote setup?*

Research Question 3.3. *How does the `task` influence the energy consumption and computation time in the local or the remote setup?*

Research Question 3.4. *Does the `format` influence the computing time and power consumption?*

Research Question 3.5. *Do the different phases show different behavior regarding the above-mentioned questions?*

Finally, it is planned to derive statistically sound decisions for the varying `cnn`s and `connectivities` whether to infer a given task locally or remotely, which leads to

Research Question 3.6. *Can it be determined with statistical confidence that remote or local inference will consume less energy or computation time on average?*

Research Question 3.7. *Is it possible to ensure the above question with respect to* `cnns` *and* `connectivity`*?*

3.3 Experimental Results

This section will discuss the evaluation that was conducted to address the research questions and achieve the research goal. The section is structured in two main parts: First, the evaluation setup will be outlined, and second, the results will be presented. The interpretation of the results with respect to the setup and the model follows in Section 3.4. Issues relevant to the validity of our study are also presented in Section 3.4.2.

3.3.1 Evaluation Setup

To provide a clear understanding of the evaluation, the technical setup will be described, along with the measurement and computation of metrics, such as computation time and energy consumption. Additionally, the statistical evaluation setup will be outlined, including which variations of parameters are considered and how many times a setup is repeated for statistical purposes.

Technical Setup

The experimental design consists of a Virtual Machine (VM) providing MoReA as an RESTFul API within a local network and an NVIDIA Jetson Nano 4GB using the API as the client. The VM has a 10,000 MBit/s ethernet connection, while the network between the client and the first router is limited to 1,000 MBit/s. The VM is equipped with 4 vCPUs, 16 GB memory, running Ubuntu 18.04, and is provided by a VMware vSphere 7 Essentials Server. The NVIDIA Jetson Nano 4GB Developer Board, which is a tiny GPU-accelerated IoT SoM, runs on the Jetson Nano Developer image based on Ubuntu. To provide a solid and stable connection to

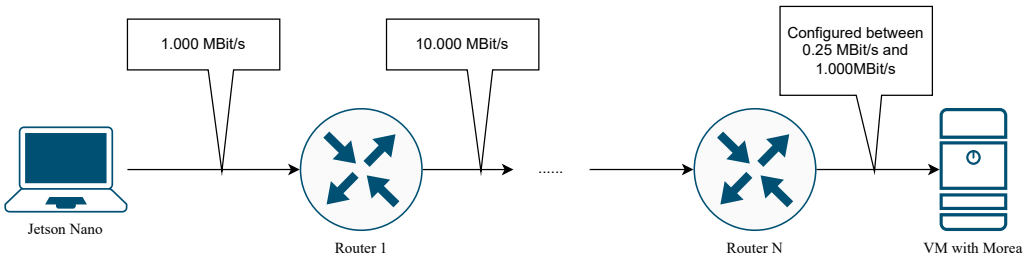


Figure 3.3: Experimental setup for MoReA

the local network, the NVIDIA Jetson Nano is connected with a Cat6a ethernet cable. Both devices are separated and located in different buildings. The VM is available for the client within three hops, and the latency is $< 1ms$. No additional devices are connected to the Jetson Nano during the entire experiment. The internal power meter that is integrated into the Jetson Nano is used for the power measurement. The measurement started just before the model's execution and stopped immediately after the model's execution.

Measurement and Computation of Metrics

Two decisive metrics are used in the aforementioned research questions: 1) Computation time and 2) energy consumption. At a given rate of 0.1 s, the current time as a date stamp, the current I [mA], and voltage U [mV] of the JetsonNano are measured. Given that energy consumption is defined as follows:

$$E = U \cdot I \cdot t \tag{3.1}$$

and power consumption is not constant throughout the measurement; consequently, the upper and lower sums for the respective time frames are calculated, i.e.,

$$E_{lower} = \sum_{i=1}^{n} U_{i-1} \cdot I_{i-1} \cdot (t_i - t_{i-1})$$

$$E_{upper} = \sum_{i=1}^{n} U_i \cdot I_i \cdot (t_i - t_{i-1})$$

In this case, $i = 1$ describes the first observation entry for either every phase (SETUP, PROCESSING, POSTPROCESSING, END) or the entire measurement, while n refers to the last observation entry of either every phase or the entire measurement.

Finally, for every phase, the average of E_{lower} and E_{upper} is reported, as well as the duration of this phase. When considering all four phases, the results are combined to obtain the final metrics.

Table 3.1: CNN models and parameters for study (a subset of the CNNs in tab. 2.1 in chapter 2).

CNN	Input Size	Layers	Neurons	Weights
densenet121	224×224	121	49,926,612	7,978,856
densenet169	224×224	169	60,094,164	14,149,480
densenet201	224×224	201	77,292,244	20,013,928
efficientnetb0	224×224	240	25,117,095	5,288,548
efficientnetb1	240×240	342	40,150,331	7,794,184
efficientnetb2	260×260	342	50,908,981	9,109,994
efficientnetb3	300×300	387	87,507,971	12,233,232
efficientnetb4	380×380	477	180,088,531	19,341,616
efficientnetb5	456×456	579	358,290,427	30,389,784
efficientnetb6	528×528	669	605,671,091	43,040,704
efficientnetb7	600×600	816	1,046,113,195	66,347,960
inceptionresnetv2	299×299	164	81,201,907	55,813,192
inceptionv3	299×299	48	32,554,387	23,817,352
mobilenet	224×224	28	16,848,248	4,231,976
mobilenetv2	224×224	53	21,815,960	3,504,872
nasnetlarge	331×331	1041	290,560,171	88,753,150
nasnetmobile	224×224	771	27,690,705	5,289,978
resnet101	224×224	101	55,886,036	44,601,832
resnet101v2	224×224	101	51,261,140	44,577,896
resnet152	224×224	152	79,067,348	60,268,520
resnet152v2	224×224	152	75,755,220	60,236,904
resnet50	224×224	50	25,612,201	210,767,874
resnet50v2	224×224	50	31,381,204	25,568,360
vgg16	224×224	16	15,262,696	138,357,544
vgg19	224×224	19	16,567,272	143,667,240
xception	299×299	71	62,981,867	22,855,952
alexnet	227×227	8	650,000	58,325,066

Table 3.2: Bandwidth and latency settings.

Equivalent Mobile Network	2G	3G	3G HSPA	4G	5G
Latecny	450ms	300ms	110ms	40ms	1ms
Bandwidth	0.25MBit/s	42.2Mbit/s		500Mbit/s	1,000MBit/s

Experimental Setup

The experimental setups stay as close as possible to the model from Section 3.2.2: There are two variations for the inferencing task: First, to compute the inferencing locally on the JetsonNano, and second, to offload the inferencing via MoReA. Two local computation options are tested, namely with an ONNX saved model (*Local-ONNX*) and a TensorFlow model (*Local-TF*) from the Tensowflow-Hub (note, however, that the download time is not counting towards the inferencing time). MoReA is employed on the described server VM for the offloading option. On this VM, the ONNX saved models are used to compare computation time. Additionally, the connection capabilities are varied to enable a trade-off decision based on task, connection, time, and power constraints. The different latencies are simulated with the Linux package TC[2] while the bandwidth is limited at the VMwares' virtual network of the VM hosting MoReA. The settings illustrated in Table 3.2 are used to emulate different mobile network standards. To see the influence of different tasks, a benchmark of 27 common CNNs is created, whose details are listed in Table 3.1. These samples contain CNNs optimized for mobile devices as well as achieving the best performance in accuracy. Table 3.1 shows that all CNNs have a different number of layers and differ in the number of neurons and weights. However, the input size (i.e., image dimension) is identical for some CNNs. The rationale for this is that many CNNs are trained on the Imangenet Dataset [64]. Overall, this yields a heterogeneous benchmark set that is well suited to evaluate the general question of local or remote execution.

This leads to $27 \cdot 2 + 27 \cdot 6 = 216$ different setups to analyze. Each setup is repeated 10 times to validate its technical accuracy and achieve statistical significance for later hypothesis tests.

[2] `https://manpages.ubuntu.com/manpages/xenial/man8/tc.8.html`

3.3.2 Result Description

This subsection describes the results of the evaluation as described above. The first part deals with descriptive results, i.e., describing what the obtained data shows and the second part deals with inductive results, i.e., what can be deducted from the obtained data for comparable situations.

Descriptive Results

First and foremost, it is necessary to validate the technical setup because it yields comparable and dependable results. To that end, the variation coefficients ($\nu = \sigma/\mu$, with μ the average and σ the standard deviation) are computed for every setup and displayed in Figure 3.4 for the energy consumption results and in Figure 3.5.

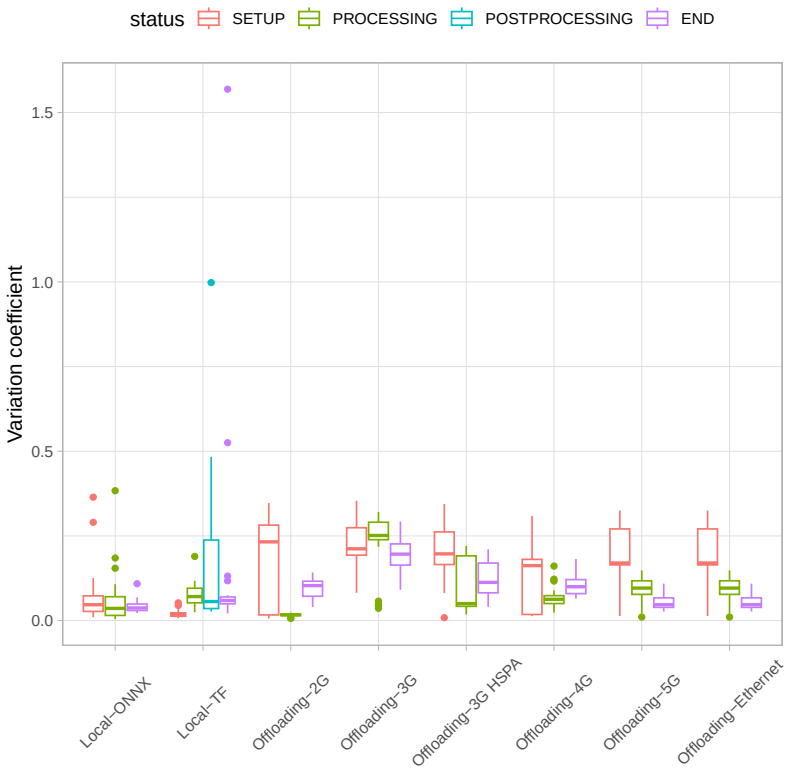


Figure 3.4: Depiction variation coefficients of energy consumption for all setups.

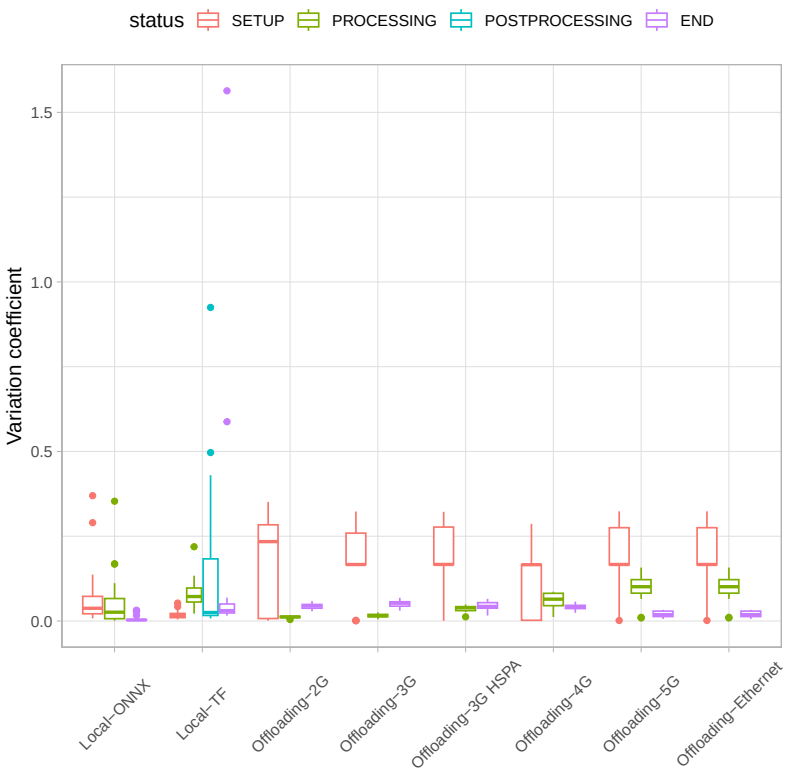


Figure 3.5: Depiction variation coefficients of energy consumption for all setups.

Both figures show the variation coefficient data separated by phase (see setup legend) and `connectivity` and `format`. The different `cnn` are shown through the boxplot depiction[3]. For example, the highest dot in Figure 3.4 represents the variation coefficient for an END-phase and the local setup with a TensorFlow format. For computation time as well as energy consumption, variation coefficients can be seen mainly between 0 and 0.5. For computation time, the SETUP-phase shows the highest instabilities, although with increasing `connectivity`, the PROCESSING phase shows increasing instability as well. For energy consumption, the SETUP-phase has higher variation coefficients as well, although, in general, there is more instability to be seen. Generally speaking, however, the results show small variation coefficients, which indicate a stable technical setup.

[3]Boxplots have their usual interpretation: Box shows the area between first and third quartile, line depicts the median. Whiskers show $\pm 1.5 \cdot IRQ$ where IRQ is the interquartile range or extremal values if smaller, dots show values outside of this range.

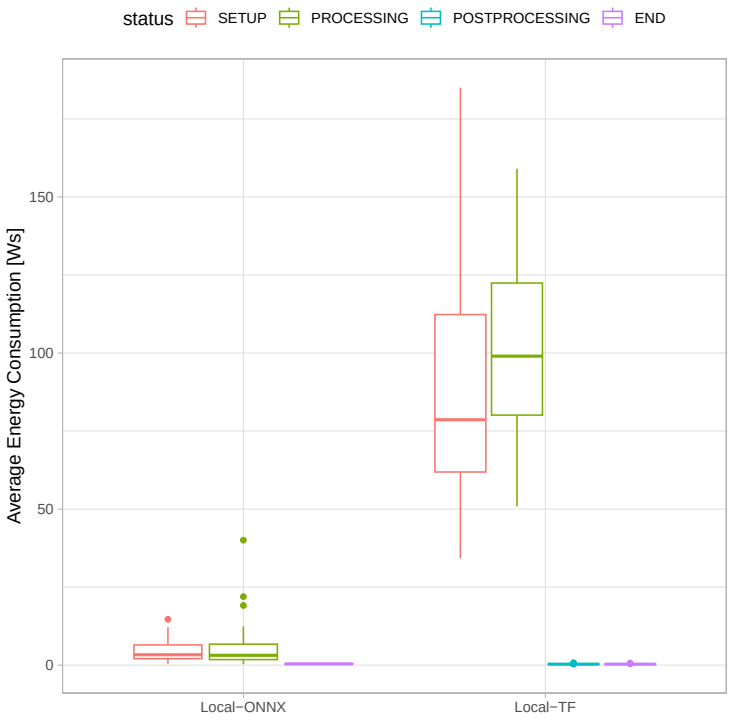


Figure 3.6: Depiction of average energy consumption for local execution with TensorFlow and ONNX formats differentiated by phases. Boxplots represent the CNN results.

The focus is on investigating computation time and energy consumption for the local setup and the influence of the `format`. Figure 3.6 shows the average energy consumption of all setups. Different colors show the four phases, and the x-axis differentiates the `format`. The resulting 27 `cnn` are shown through the respective boxplots.

The comparison between the ONNX-formal and the TF-format shows a large difference in the results for the SETUP- and PROCESSING-Phase. In both cases, the energy consumption is higher for the TF-format than for the ONNX-format. The missing boxplot POSTPROCESSING indicates that this must have been shorter than the measurement distance of $0.1s$. The same holds for computation time, although not depicted here. To have a more balanced comparison, the analysis only focuses on the ONNX format for comparing remote and local execution.

Comparing the local setup (with `format`=ONNX) with the remote setup (with

varied connectivities) results in the similarly structured Figures 3.8 and 3.7. In this case, both measured metrics, computation time and energy consumption, will be presented and explained. Figure 3.8 illustrates that the POSTPROCESSING and END-phase barely have an impact on the overall energy consumption. For the remote setups, the SETUP-phase also has a negligible influence. The PROCESSING-phase makes up most of the energy consumption. This is different for the local setup: Here, the SETUP and PROCESSING-phase have roughly the same influence on the overall energy consumption. Comparing only the PROCESSING-phases for the local and the remote setups, it can be seen that the slower connectivities, i.e., 2G and 3G, seem to induce a (in general) higher energy consumption than the local PROCESSING-phase. From 3G-HSPA onward, however, the energy consumption is smaller for most CNNs. Additionally, it is noticeable that the cnns induce small to no variance to the energy consumption values for the remote setups; however, there is a much higher degree of variance for the local setup.

The computation shows a similar behavior regarding the influence of the different phases. However, the results for the PROCESSING differ for the remote setups. Setups with slower connectivities show higher computation times for almost all CNNs (the boxplots lie higher than for the local setup). Only connectivities from 4G onwards are faster during the PROCESSING-phase. The slower the CONNECTIVITY, the higher the variance of CNN results in the setup. Comparing this to the variation coefficient analysis from above, however, it seems that this is only an absolute increase in variance but not a relative one. To get an overview of the total computation time and total energy consumption, a comparison of local and remote setups is presented in Figure 3.9. Here, each point depicts the averaged results for one setup (total average computation time on the x-axis and total average energy consumption on the y-axis). Equal colors show equal connectivities, while the ellipses are a visual aid for the reader.

Three main patterns can be seen in Figure 3.9: (a) The different connectivities show a linear increase in energy consumption as well as computation time, with slower connectivity producing higher values for both metrics. Nevertheless, the increase is comparatively higher for computation time than for energy consumption. (b) The local setup seems to follow a linear trajectory with a signifi-

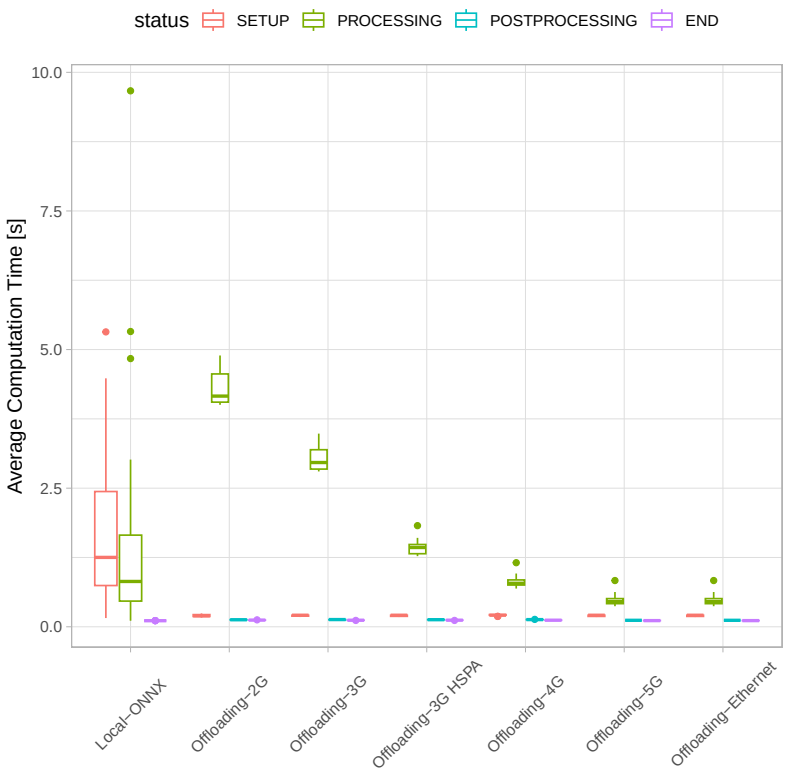


Figure 3.7: Depiction of average computation time for local execution with ONNX and remote execution with different connectivities. Boxplots represent the CNN results.

cantly higher slope. (c) The total values show that the total energy consumption is (generally) lower for the four fastest `connectivities`, and the total computation time is mostly higher in the remote setups than in the local setup.

The local setup shows high variances for the CNNs especially; therefore, a closer look into the influence of CNN parameters is important. As complexity measures for the CNNs, the following parameters are considered: weights, layers, and neurons. Additionally, the actual `task` is considered, i.e., the image size to be inferenced. In the understanding of this analysis of the offloading process presented in Section 3.2.2, an influence of the `task` is assumed.

Figure 3.10 shows the average energy consumption for the SETUP and PROCESSING-phase in different setups in dependence of the number of weights in a CNN. Colors show different setups, i.e., remote or local execution and different

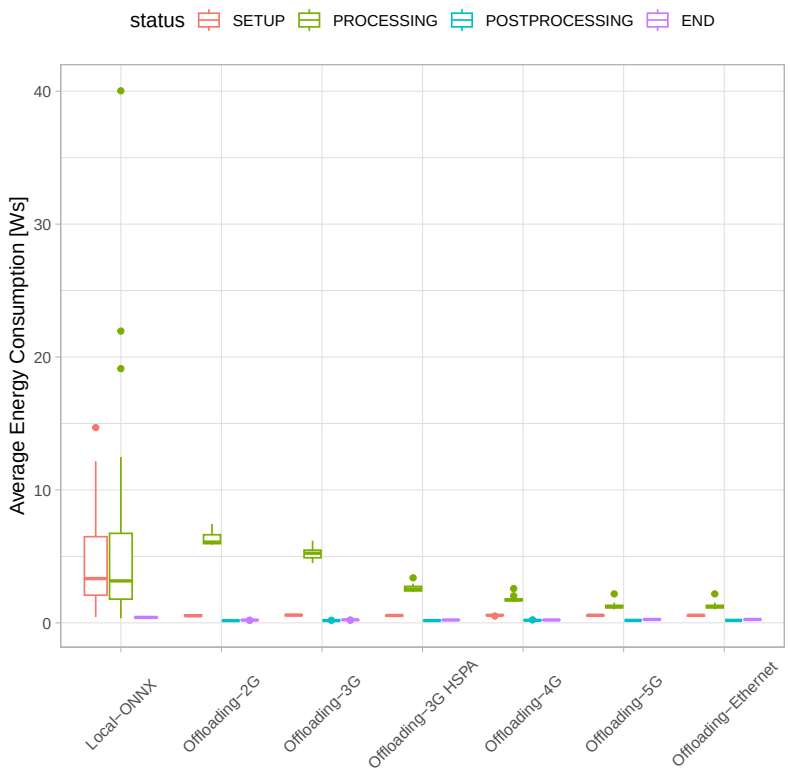


Figure 3.8: Depiction of average energy consumption for local execution with ONNX and remote execution with different connectivities. Boxplots represent the CNN results.

`connectivities`, while the shape shows the phase the results stem from. The focus is on SETUP and PROCESSING because the third and fourth phases have a negligible effect on the overall result and barely differ in their behavior comparing local and remote execution. It can be seen that the remote setups seem not to be influenced by the weights of the CNNs. This is also the case for layers and neurons, as well as the size of the inferencing task (not shown as a figure here). Equivalently, when zooming in on the smaller values, the results seem not to be affected by the considered parameters.

The local setup, nevertheless, shows a somewhat increasing behavior for both phases. Analyzing this in close-up yields the results presented in Figures 3.11 and 3.12. Figure 3.11 shows the average energy consumption in the local setup for the SETUP-phase. The energy consumption results seem to show an almost

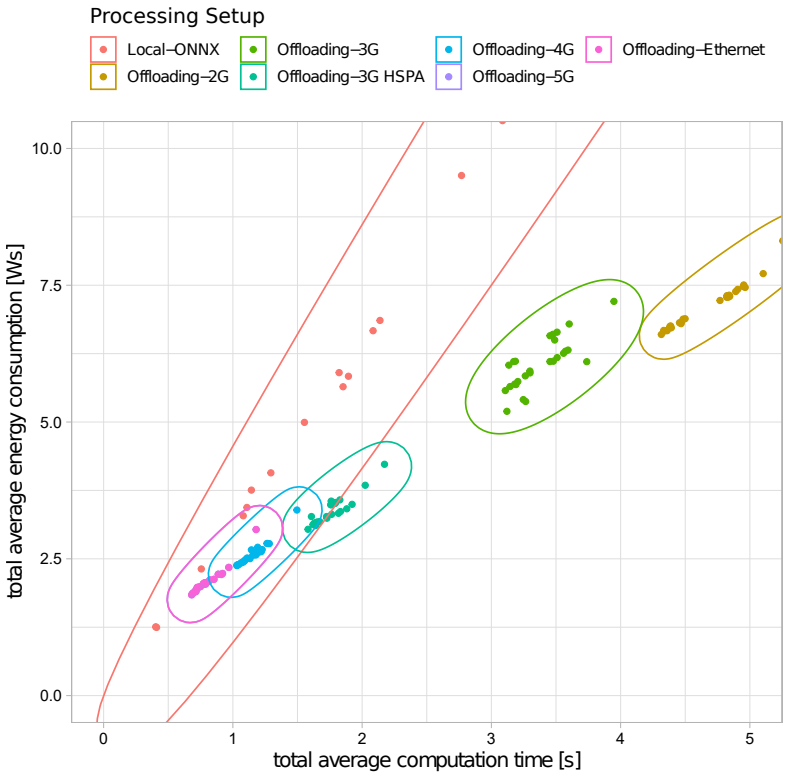


Figure 3.9: Depiction of total average energy consumption and total computation time per CNN, differentiated by evaluation setup.

linear behavior in the weights except for one outlier, the CNN, with the highest number of weights. However, when trying to verify an identical influence for the PROCESSING-phase, the results were disparate. Factoring in a possible quadratic influence of the image size as the measure for the `task` size, however, Figure 3.12, which shows the ratio of the average energy consumption and the squared image size in dependence of the number of weights in the CNN for the PROCESSING-phase of the local setup. In this case, it can be seen as a comparable linear relation except for the same outlier as before.

Inductive Results

This section is aimed toward reliable generalized statements in which setting offloading is to be preferred. To that end, the first hypothesis tests are performed

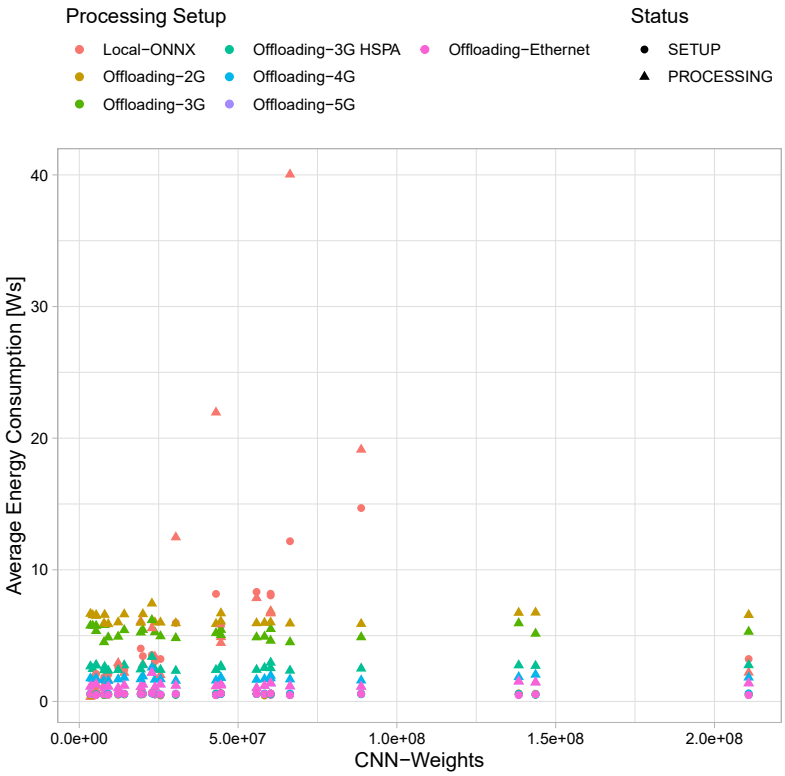


Figure 3.10: Depiction of the number of trainable weights of the CNN with energy consumption for SETUP- and PROCESSING-phases for different evaluation setups.

for energy consumption and computation time for each `cnn` and `connectivity` with the local setup as the null hypothesis. The hypothesis test setup is as follows: a Welsh's t-test is performed with a maximum error probability of 0.0001 for each CNN, comparing the different remote setups to the local ONNX one [91]. The Welch-Satterthwaite-Equation approximates the respective degrees of freedom [92].

A hypothesis test is counted as *won* for the remote setup if the t-value exceeds the 0.9999-quantile of the t-distribution with the respective degrees of freedom. Usually, it would be required to perform a Holm-Bonferroni family error correction to obtain statements about all CNNs however as a very small error probability is selected for the singular hypothesis, the familywise error still holds at $\alpha_{all} = 1 - (1 - \alpha)^{\#CNNs} = 1 - (0.9999)^{27} \approx 0.0027 < 0.01$. Hence, all statements can be

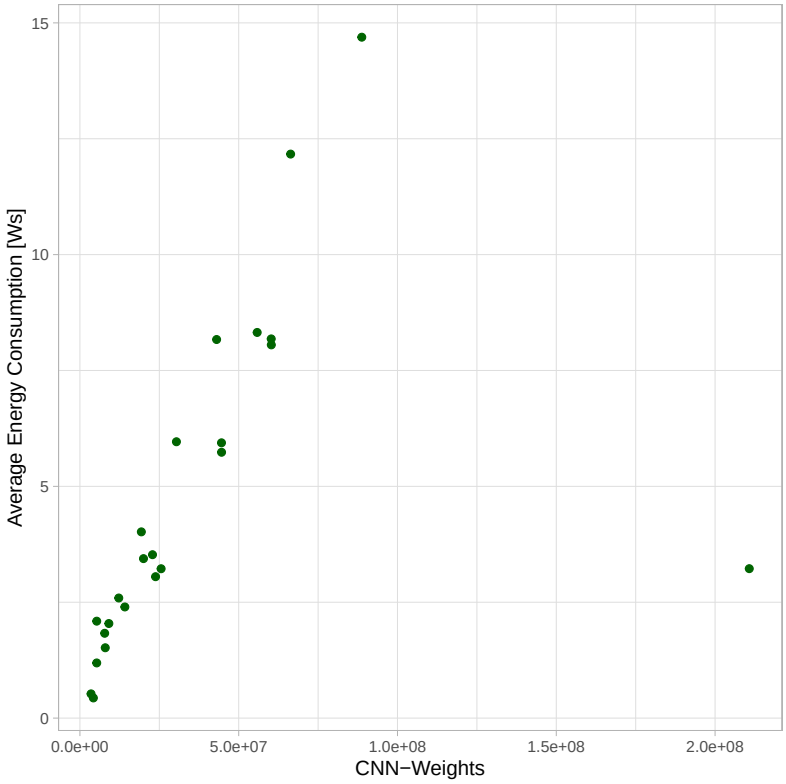


Figure 3.11: Depiction of the number of trainable weights of the CNN with the energy consumption ratio to image size for SETUP-phase for the local setup with ONNX format.

made with an error probability of less than 1% [93].

The number of won hypothesis tests are reported in Figure 3.13. The left side shows the results for energy consumption, while the right side shows the results for the computation time. The black line depicts the maximum number of tests that could have been won, i.e., the number of CNNs. So, e.g., the left-most column can be read as for 11 CNNs, the average energy consumption of the remote setup with a 2G connectivity is smaller than the average energy consumption of the local setup (with ONNX) with an error probability of less than 1%. It can be seen that the number of won tests increases with improving `connectivity` for energy consumption as well as computation time. In both cases, the best result is at 21 of 27 CNNs. Computation time has less won tests for 2G and 3G connectivities but gains significantly with 3G-HSPA.

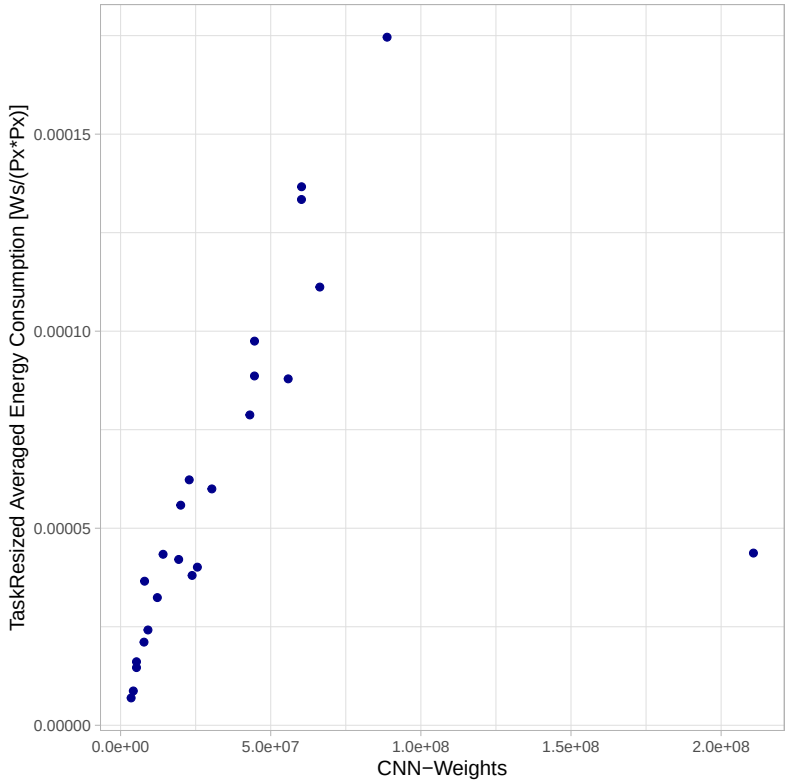


Figure 3.12: Depiction of the number of trainable weights of the CNN with the energy consumption ratio to image size for PROCESSING-phases for the local setup with ONNX format.

Figure 3.14 now relates the information of won hypothesis tests to the CNN metric that was found to be partially explanatory for energy consumption and computation time: The number of weights, i.e., trainable parameters. However, the t-value depicts results from the hypothesis test on the y-axis. The line in the figure depicts the 0.9999 quantile with $df = 4$[4] So, for each point representing a hypothesis test for a CNN that is above the solid black line, the hypothesis is won. The colored lines show the overall linear trend. Generally, it is visible that with increasing weight, the hypothesis tests are more likely to be won, i.e., the remote setup has a smaller computation time or energy consumption. For energy consumption, the setups with better connectivities (4G, 5G) almost entirely lie

[4]Officially, the 0.9999 quantile needs to be computed with different degrees of freedom; however, in all our data, the degrees of freedom never were smaller than $df = 4$

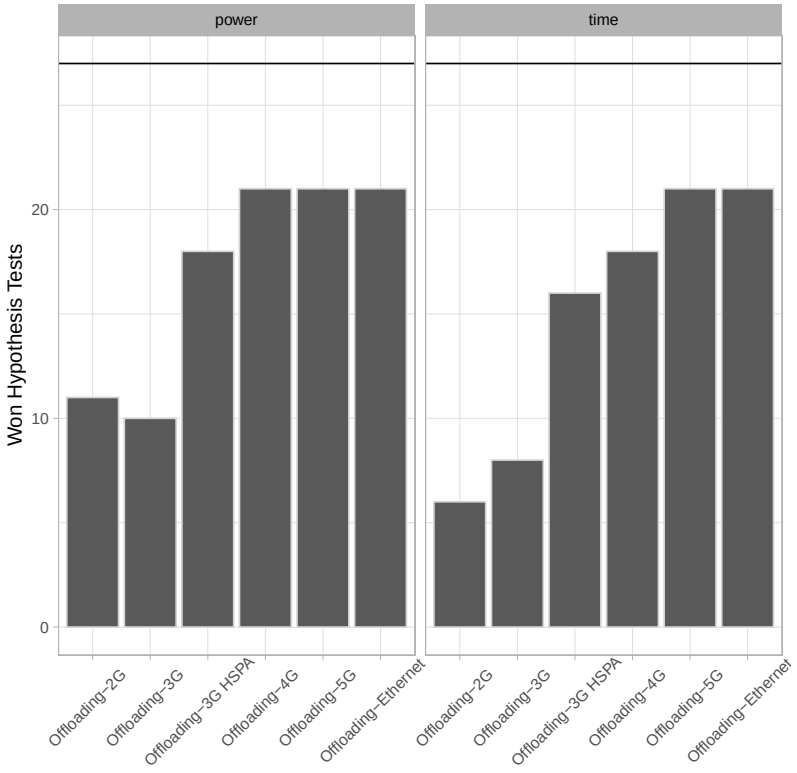


Figure 3.13: Number of won hypothesis tests by remote setup connectivity.

above the quantile line, and the setups with worse connectivities catch up from about $2.5 \cdot 10^7$ weights. For computation time, the picture is different. 2G and 3G setups only perform faster than the local setup for CNNs with a number weights of $4 \cdot 10^7$ weights. The slope of the slower connectivities seems to be higher, i.e., increasing the number of weights leads to a more rapidly increasing probability of winning the hypothesis test.

3.4 Discussion

This section discusses the evaluation results from Section 3.3.2, answers the proposed research questions, and describes threats to the validity of our study.

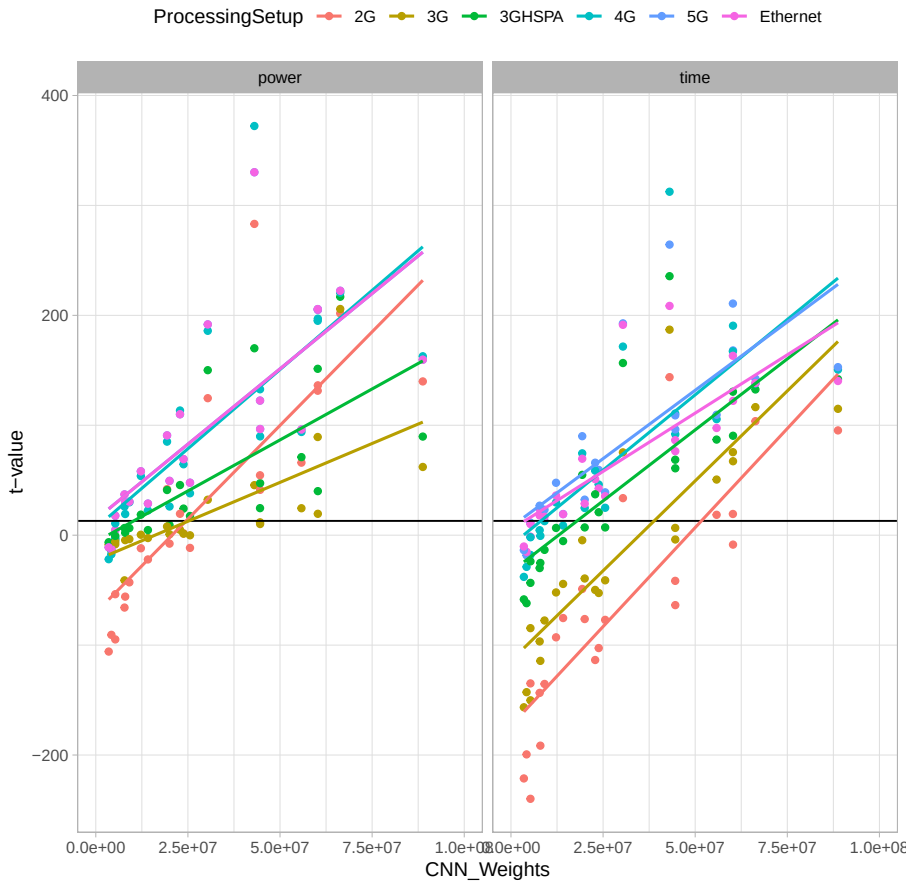


Figure 3.14: Depiction of t-statistic value for individual CNNs against their weights, colors depicting different connectivities.

3.4.1 Evaluation interpretation

The results are discussed for the research question (research question 3.5 is answered within research question 3.1 to 3.4, and research question 3.6 and 3.7 are answered together)

Research Question 3.1: Connectivity Evaluation results have shown that the `connectivity` is the main factor for the energy consumption and computation time results when offloading. While energy consumption already decreases for slow connectivity, computation time only shows advantages for fast connectivity.

Research Question 3.2: CNN CNN-parameter (weights) do have a linear effect on the SETUP phase and partial effect on the PROCESSING phase, however, only

in the local setting. The remote setting does not seem to be influenced by its parameters. The interesting factor here is computation time, as it linearly influences the energy consumption for the remote setting because the power consumption of the local machine is constant during the offloading phase (it is idle). For the computation time, it holds that the slower the connectivity, the smaller the influence of the actual inferencing and, thus, the CNN parameters. The remote machine seems to be so performant that the CNN parameters result in no measurable effect.

Research Question 3.3: Task For the remote setup, there is no measurable influence, same as for the research question 3.2. For the local execution, however, it is plain to see that only the PROCESSING Phase is influenced by the inferencing task, not the SETUP phase. So, at least in our setting, the task preparation has no measurable effect.

Research Question 3.4: Format When comparing the TensorFlow format to the ONNX format on the local machine, the evaluation shows huge differences. The TensorFlow form takes up significantly more time and energy than the regarded cases.

Research Question 3.6 and 3.7: Remote vs. Local First and foremost, our results show that if energy saving is more important than timing issues, offloading is advisable for about 40% of the considered CNNs, even with the worst connectivity. When improving the connectivity, this percentage improves 78%, and the trade-off eases (and eventually vanishes), i.e.for, 5G offloading does not have worse effects on computation time than on energy consumption (compare the number of won hypothesis tests in Figure 3.13). Generally, the more complex the CNNs (in terms of weights) and the better the connectivity, the higher the improvement in both energy consumption and computation time. When having a 3G-HSPA connectivity, offloading almost always makes sense in terms of energy consumption savings. It improves computation times as well for CNNs with at least 25 million trainable parameters (weights).

Following this discussion, Table 3.3 summarises our findings, fulfilling the research goal. It shows decision support for offloading based on connectivity and CNN weights. The first advisement is for a focus on energy saving, the second for time (*l* refers to local, *o* to offload).

Table 3.3: Summarised decision support

Connectivity/ CNN	slow (2G, 3G)	medium (3G-HSPA)	fast (4G/5G)
low (< 25 Mill)	l/l	o/l	o/o
medium (< 50 Mill)	o/l	o/o	o/o
high (> 50 Mill)	o/l	o/o	o/o

3.4.2 Threats to validity

As with every modeling approach and experimental study, there are some design decisions or outcomes that threaten the validity of the approach. The threats of this chapter are stated here:

Lacking Variations Our experimental study misses variations on some of the identified influencing parameters, namely the performance of the local and remote machines, the type of task, and the number of tasks to be performed simultaneously. It is decided not to vary the performance of the machines as in the actual real-world scenario, where the decision maker most likely has no option to change either the local setup or the remote setup. Nevertheless, varying these might yield a broader picture of the interdependencies at play. The task type was not varied because the scenario involves image inferencing on CNNs. Therefore, the scope was restricted to this scenario. Varying the number of tasks could shed some light on the influence of the setup phase and the duration of the processing phase. It is planned to investigate this in further research, together with the machine's performance. Additionally, after seeing the results for the local setup of the TensorFlow format, it is decided to stick with the ONNX format for the remote case. Employing the TensorFlow format remotely as well and investigating other formats again remains a task for the future.

Missing parameter information Good results are achieved using the ONNX format when analyzing the CNN parameter influence on the different phases. However, the behavior of the TensorFlow format cannot be fully explained. The data analysis shows that there seems to be another influencing factor that we did not observe. Identifying this factor will be another task for further research but might require a different experimental setup.

Used tooling The bandwidth and latency are simulated using VMware virtual

network settings and the Linux tool TC, respectively. However, the behavior on a real-world wireless connection may differ from that in our experimental setups. To minimize this bias, a controllable experimental setup is designed to reproduce the results as closely as possible to real-world scenarios. Still, there may be some discrepancy between the simulation and the real world.

3.5 Conclusion

This chapter introduces a detailed energy and performance analysis for CNN inferencing tasks on local execution on IoT devices and offloading to edge or cloud systems with different mobile networks. The analysis shows that the CNN weight, bandwidth, and latency are significant influencing factors. Based on the experimental results, a detailed guide is created to advise when to offload and when not. The experimental results illustrate that, in most cases, offloading is reasonable for power-saving purposes. Higher bandwidths are required for performance purposes, and slow networks like 2G and 3G can become the bottleneck in applications' performance.

In future work, to extend the analysis setup, it is planned to analyze the energy-time trade-off in real-world scenarios with local small- to medium-sized businesses and different machines.

4 Power Estimation of Convolutional Neural Networks on GPGPUs

This chapter presents a novel approach to estimate the power consumption of CUDA-based CNNs on GPGPUs in the early design phases. The proposed approach uses a hybrid technique where static analysis is used for feature extraction and different ML regression models are utilized for power estimation model generation. However, the difference between the techniques is negligible with a MAPE of 8.8 % and 8.4 %, for NN and K-NN, respectively. As the K-NN is more accessible to build, train, and use, it illustrates the advantage of shallow learning methods compared to DNN. Experimental results demonstrate that the proposed approach based on K-NN can predict CNNs power consumption up to a *Absolute Percentage Error* of 0.0003% compared to the real hardware. The introduced research is based on published material from [5, 6, 21].

The chapter starts with a brief introduction to power estimation of CNN on GPGPUs in Section 4.1, followed by the methodology in Section 4.2. Afterward, Section 4.3 presents the experimental results, followed by a discussion in Section 4.4. Finally, the chapter closes with a short conclusion in Section 4.5.

4.1 Introduction

The number of IoT devices that leverage ML algorithms has considerably increased in the last decade, ranging from manufacturing to scientific- health- and security-related applications [94]. Among the existing ML algorithms, CNN is

widely used in pattern recognition tasks and image analysis because it can handle large and unstructured data [10, 95].

One of the significant challenges (see chapter 1 challenge 3) that designers commonly face during the design phase of such IoT devices is to choose the right ML accelerator that adheres to the design constraints such as low power consumption, latency, and cost of the final products [5, 21]. For example, assume that designers need to design an IoT device where its CNN application is performed on a GPGPU (as hardware accelerator). In the case that the power consumption and battery lifetime of the IoT device are considered as the design constraints, choosing the most proper GPGPU early in the design phase can significantly avoid costly design loops occurring as fewer prototypes need to be built. Moreover, in the case of Cloud-based IoT devices where data processing of the CNN application performs remotely on Cloud-based accelerators (i.e., GPGPUs), choosing an appropriate GPGPU can significantly reduce the renting cost, resulting in a direct impact on the price of the final product.

Power estimation techniques are a promising solution to this issue. A robust power estimation approach enables designers to choose the most appropriate GPGPU that meets the constraints, early in the design phase. To estimate power consumption, existing methods mainly rely on so-called performance counters [22, 37, 38]. Consequently, their estimation depends on the run-time data, meaning the ML model must be run once on the target GPGPU so that the performance counter results can be measured. However, this can limit the usage of such methods in the early design phase as the GPGPU must already be selected. Moreover, this can increase the required analysis time.

This chapter focuses on the power estimation of CNNs on GPGPUs, which is one of the most popular ML algorithms in automated manufacturing [1]. A novel approach is presented, enabling designers to predict the power consumption of a given CNN even with small training datasets in the early design phases. The proposed approach uses a hybrid technique where static analysis is used for feature extraction, and the ML regression analysis is utilized for power estimation model generation. The static analysis for features extraction is performed on PTX code (which is generated at compile time) of CNNs, GPGPUs' architectural information, and CNNs topology. To create the power estimation model, K-NN as lightway ML

approach as well as NN are used to learn the power consumption behavior of CNN's inferencing on GPGPUs. Afterward, the results of both techniques are compared to each other.

Unlike the existing methods that use performance counters for their prediction, the proposed approach takes advantage of PTX code (which is generated at compile time) and GPGPUs' architectural information. Both PTX code and GPGPUs' architectural information are easy to collect and make our approach easy to apply and transfer to other CUDA-based applications. This enables the designer to choose the most efficient GPGPU in terms of power consumption among the existing models at compile time. Experimental results illustrate the effectiveness of the proposed approach in estimating the power consumption of CNNs on GPGPUs where up to a MAPE of 8.8% and 8.4% for NN and K-NN, respectively, in comparison to the real hardware execution is achieved.

4.2 Methodology

Focusing on the available information is essential to estimate the power consumption of CNNs in the early design stage. The early available information that does not rely on CNNs execution on real devices is 1) the GPGPUs' architectural details that are available for the different GPGPUs, 2) the low-level PTX code that is generated at compile time, and 3) CNN architecture and topology like its trainable parameters. In contrast to [22, 38], only features from the sources above are considered for the power estimation and do not rely on the performance counters, which are only available at run-time. Performance counters would require at least one execution on an actual device to collect them. This can limit the usage of such practices in the early design stage as the target GPGPU must already be selected.

The proposed methodology is illustrated in Fig. 4.1, which has three main phases: 1) information extraction, 2) training dataset creation, and 3) predictive model generation.

In the first phase, different CNNs are selected, and how they load various components of GPGPUs is analyzed. The architectural information (e.g., CUDA

Cores, Memory, or L2 Cache) which are available for different series of GPGPUs are compared between different GPGPU models. By this, those GPGPUs' attributes and components that impact performing CNN models are extracted. Next, the CNNs are compiled to PTX, and the PTX code for each CNN is analyzed. The instructions loaded into the GPGPU are extracted, and classes of instructions are built. Each class contains the number of instructions in the PTX code for a CNN. Also, a high-level CNN analysis is performed and extracts the number of trainable parameters for each CNN.

In the second phase, a training dataset is built where the classified extracted CNN instructions and the GPGPU components (that have an impact on performing CNN models) are considered as inputs. The amount of power consumption for each CNN running on the GPGPU as output. The amount of power consumption for each CNN is measured on three different NVIDIA GPGPUs (K80, 1080Ti, and V100S) with the nvidia-smi tool. Since the ML regression algorithms are sensitive to the selected features [96], the right combination of features needs to be detected. Thus, instead of using all extracted data as features, several combinations are generated to search for the most relevant features. Having fewer features leads to simpler models that require shorter training time, reduce the chance of overfitting, and are easier to interpret.

Next, in the third phase, the ML regression algorithms are applied to the generated training dataset for power estimation model generation. Once the predictive models are trained, they can be used to estimate the power consumption for a given CNN on different GPGPU architectures. Finally, during the model evaluation, the best-performing predictive model is selected.

4.2.1 GPGPUs Architecture Analysis

The power consumption of GPGPUs is affected by many factors. NVIDIA lists all the GPGPU's architectural details in [69, 97]. All architectural details for each GPGPU have been extracted for the experimental setup. The values for the components are transformed into comparable measurement units to ensure the ML method gets the feature for different GPGPUs in the same unit. Moreover, several CNNs are executed on three different GPGPUs and monitor the component uti-

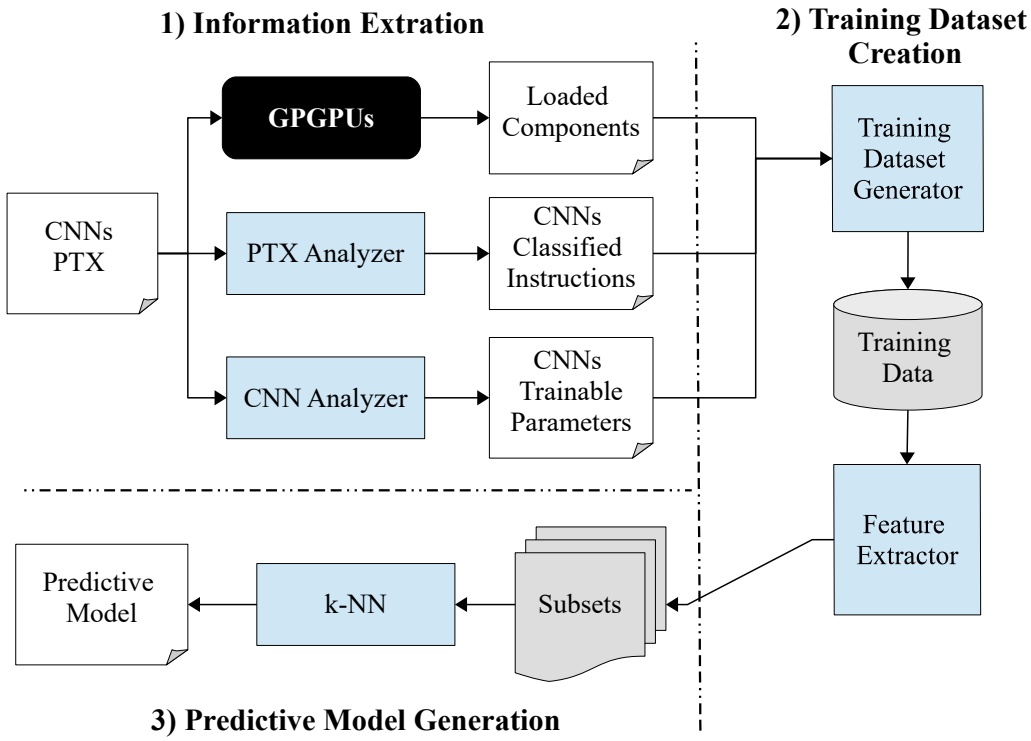


Figure 4.1: Overview of the proposed methodology.

lization. This gives more precise details of which component is affected by CNNs execution and is involved in the power consumption. Furthermore, GPGPUs architectural attributes are considered, namely: maximum temperature, transistor size, or power supply. This gives more differentiation between the various GPGPUs.

4.2.2 PTX Instructions Analysis

NVIDIA provides designers with the CUDA Library to develop GPGPU applications and to write proper code for GPGPUs programming. NVIDIAGPGPUs run so-called kernels. Each kernel is a set of PTX instructions generated by compiling the CUDA code with NVIDIA CUDA Compiler (nvcc) compiler. The PTX code is a stable low-level programming model and ISA for general-purpose parallel programming. Fig. 2.3 (see chapter 2.4) shows a part of a given CNN's PTX code. The PTX code is given to the GPGPU driver and interpreted at run-time [98]. The

PTX code contains detailed information of all memory accesses (read or write) and computational instructions that will be executed on the GPGPU.

For a given CNN, a static analysis is performed on its corresponding PTX file to count the number of appearances of the instruction that impacts the power consumption of the CNN when it runs on a GPGPU. Moreover, the number of threads started for executing the PTX code are read out. The number for each instruction is multiplied by the number of threads to consider the effect of threads on CNN's power consumption when it runs on the GPGPU. The result of this analysis is stored in the *CNNs Classified Instructions*, including a set of classes (each instruction is associated with a single class) and the number of calls in the PTX code. Hence, not all existing PTX instructions are used for power estimation model generation. Instead, only those instructions are considered that appear in the PTX code of CNN benchmarks and directly impact power consumption. As illustrated in Fig. 4.1 phase 1, the static analysis is performed by *PTX Analyzer* module for several PTX files from different CNN algorithms. The results of this analysis are used to create the training dataset in the next step.

4.2.3 CNN Topology Analysis

Every CNN can be characterized based on its architecture and topology. The main feature of this characterization is the CNN's trainable parameters. Trainable parameters change during the training and are an essential aspect of CNNs. These parameters are the weighted connections between neurons, and their number varies across different CNNs. A CNN's number of trainable parameters reflects its complexity and computational operations. Therefore, by looking at the number of trainable parameters, one can distinguish one CNN from another and add more differences to each CNN's training dataset to make it unique. It is important to note that the number of trainable parameters is the same across different GPGPUs and depends solely on the CNN. This feature makes the trainable parameters an excellent tool for introducing more differences into the training dataset of each CNN.

The proposed analysis takes advantage of Tensorflow[1] and TensorflowHub[2] to use pre-trained CNNs from the Hubs and also to implement user-defined CNNs. The Tensorflow API allows access to the trainable parameters for each CNN. As illustrated in Fig. 4.1 phase 1, the analysis is performed by *CNN Analyzer* module for all CNN benchmarks. The results of this analysis are stored in the *CNNs Trainable Parameters* and used to create the training dataset in the next step.

4.2.4 Creating Training Dataset and Predictive Model

To build the prediction model, it is vital to have a robust training dataset. Therefore, the extracted information from the first phase is used to create the training dataset D based on the following definition:

$$D = \{d_i | d_i = \{y_i, (p_i, c_i, t_i)\}; 1 \leq i \leq n\} \tag{4.1}$$

Where the parameters p, c, t are considered as inputs (the predictors) of the training dataset and denote the classified extracted CNN instructions, the GPGPU components (that have an impact on performing CNN models), and the CNN trainable parameters, respectively, the parameter y indicates the measured power consumption for each CNN running on the GPGPU and is considered the training dataset's output (the response). Each pair of predictors and the corresponding response is regarded as one observation that depicts with parameter d.

The generated CNNs *instruction Profile* and GPGPUs' architectural details are used to build the training data set. Moreover, the CNNs are executed on three different GPGPUs while the power consumption is measured with the nvidia-smi tool. The measured power consumption also goes into the training data set.

To give an overview of the training dataset D, Table 4.1 demonstrates a part of its structure. The *CNNs Classified Instructions* column lists a part of instruction classes and, for each class, the number of extracted instructions. Column *GPGPU Components* shows some relevant architectural components. Finally,

[1] `https://www.tensorflow.org/`
[2] `https://www.tensorflow.org/hub`

Table 4.1: Example of training dataset structure used to create the predictive model

Observation	CNNs Classified Instructions				GPGPU Components			Trainable Param.	Power Consumption
	Data mov & conv	FP	..	Class N	CUDA Cores	...	SM		(Output)
CNN_1 on $GPGPU_1$	8	3	...	..	5120	...	80	25549352	Power_1
CNN_1 on $GPGPU_2$	8	3	...	..	4352	...	68	25549352	Power_2
CNN_1 on $GPGPU_3$	8	3	...	..	3584	...	28	25549352	Power_3
CNN_n on $GPGPU_m$	...	...	...	..	...	...	...	...	Power_{n*m}

the *CNNs Trainable Parameters* column depicts the training dataset's last input (predictor). All three, instruction Profile, GPGPU components, and Trainable Parameter become our machine learning method's input features (predictors). The last column shows the power consumption measured by executing the PTX code of each CNN on a real GPGPU. Thus, each row of the table indicates an observation d in the training dataset D where the first three columns are the predictors (or features) while the total power consumption (column *Output*) is the response. The training dataset is split into 70% for the training phase and 30% for the validation phase.

Feature Selection

Features that exert little impact on the estimation model should be eliminated to reduce the dimensionality of the training dataset. Reducing the number of features leads to simpler models that require shorter training time, decrease the chance of overfitting, and are easier to interpret. For instance, the K-NN algorithm (used in this work for predictive model creation) is sensitive to redundant or irrelevant features [96]. To find the best feature combination and eliminate outside features, different subsets are built out of the primary training dataset D where $T_s \subset D$. The combinations start with nine different predictors and define the following three types of subsets based on them by considering:

- nine predictors out of all possible ones

$$T_{s_1} = \{d_i | d_i = \{y_i, (c_i, p_i, t_i)\}; 1 \leq i \leq n\} \tag{4.2}$$

- two subsets where at least one predictor from c must be included:

$$T_{s_2} = \{d_i | d_i = \{y_i, (c_i, p_i)\}; 1 \leq i \leq n\} \ \ or$$
$$T_{s_2} = \{d_i | d_i = \{y_i, (c_i, t_i)\}; 1 \leq i \leq n\} \tag{4.3}$$

- a combination of predictors from p and t where

$$T_{s_3} = \{d_i | d_i = \{y_i, (t_i, p_i)\}; 1 \leq i \leq n\} \tag{4.4}$$

Since the *GPGPUs' components* predictor has a lot of redundancy with little changes in the training dataset, the statistical automatic feature selection techniques based on variance cannot be used for feature selection due to confusion. The main reason is that the *GPGPUs' components* predictor marks as a low-impact predictor by the statistical feature selection. Hence, the features are selected manually to solve this issue. The experimental results in Section 4.3 confirm the importance of the *GPGPUs' components* predictor in creating the best predictive model.

Those combinations are skipped that are only including GPGPUs' architectural information and not any PTX instructions nor CNN's architectural attributes. The model would predict the same power consumption for every CNN if only GPGPU architectural features were considered. There is no change in the input values when only architectural components are considered. Consequently, there must be at least one PTX instruction Class as a feature included in the combinations. Moreover, a script is developed to generate the possible combinations and run them in parallel on our machines. The best results with a MAPE lower than 15% are saved to a log file.

K-Nearest Neighbors

To predict the power consumption of a given CNN, K-NN regression is applied to different subsets T_s of the training dataset D in the definition (4.1). K-NN is a nonparametric clustering algorithm. It can be used to perform regression prediction by calculating the average value of the k nearest neighbors' values

where Y denotes the new predicted output and y_i is the output of the ith nearest neighbor.

$$Y = \frac{1}{k} \sum_{i=1}^{k} y_i \tag{4.5}$$

To find the k nearest neighbors, a distance metric is applied to all elements in the training dataset and the new element whose value is to predict. Consequently, the run-time and complexity are linear scaling with the number of elements in the training dataset. K-NN can be used with metrics like Euclidean, squared Euclidean City-block, and Chebychev [96]. The Euclidean Distance is used, it is defined as follows: d between two vectors q and p is defined as follows:

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{4.6}$$

Finding the right K

One of the main challenges of applying the K-NN algorithm to a prediction training dataset is finding the right k. While a large value for k can smoothen the prediction and be assisted with noisy data, a small value of k can corrupt the estimation model. To overcome this issue, several experiments are performed by running K-NN for each feature combination, with k ranging from one to 20. Since there is no considerable improvement achieved by k values larger than ten, the maximum value is set to 10; k to 20 (see whole experiments in Section 4.3). The K-NN estimates the power consumption by calculating the average power consumption of the k nearest data point in the training dataset.

After running the different combinations, the best results are found in the log file, and the final model is based on the best feature combination. Therefore, the obtained predictive model can be used to estimate the power consumption of CUDA-based CNNs on GPGPUs.

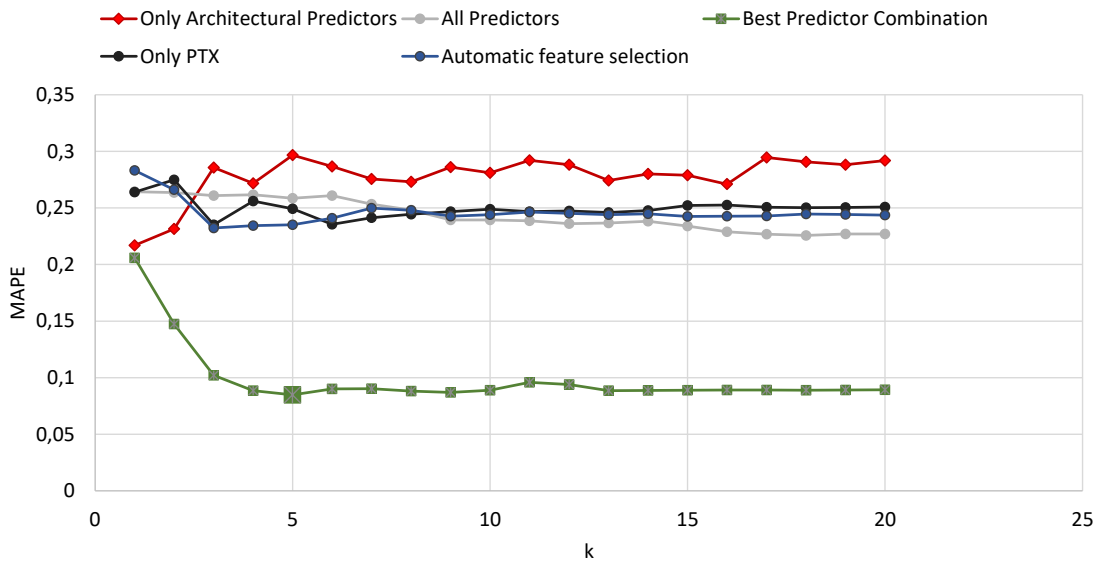


Figure 4.2: MAPE value for different k ranging from one to 20.

4.3 Experimental Results

This chapter aims to answer the following research questions to evaluate our predictive models.

Research Question 4.1. *Which ML algorithm performs best for power estimation?*

Research Question 4.2. *Which features most impact power estimation on GPGPU?*

Research Question 4.3. *In the case of K-Nearest Neighbors, What is the most appropriate k to select?*

Since the quality of the predictive model is significantly related to the robust training dataset, five different subsets of the training dataset were created based on the selection of various features (predictors) combinations explained in Section 4.2.4. To evaluate the quality of the generated predictive model for each subset of the training dataset, three different standard metrics were applied, which are 1) Root Mean Squared Error (RMSE), 2) MAPE, and 3) R^2. The lower value for RMSE indicates a better estimation model. The values of MAPE are between zero and one, where the value one stands for an error of 100%. R^2 is a value

between zero and one, where one stands for a high correlation between model and data, while a negative value indicates no correlation.

Different k values on the predictive model generation are considered when applying the K-NN regression algorithm to each subset of the training dataset. Fig. 4.2 illustrates the MAPE for different k values for $1 \leq k \leq 20$. As there is no considerable improvement by k values larger than 10, the experiments are only performed by values up to 20. Please note that in our experiments, $k = 1$ is not considered as it would mean that the predicted power consumption equals the nearest neighbor. To apply the equation from Section 4.2.4, at least $k \geq 2$ is required.

Table 4.2 demonstrates the experimental results of our analysis for each subset of the training dataset for the best value of k. Column *Training dataset subsets* lists the combination of the features that were used to create the best predictive model, which are 1) *Automatic feature selection*, 2) *All predictors*, 3) *Only GPGPUs' architectural predictors*, 4) *Only PTX predictors*, and 5) *Best predictor combination*. Please note that, for the case of *Automatic feature selection* subset, different automatic feature selection methods were used where the best result is shown in the table that belongs to the F-statistic automatic feature selection method.

As illustrated in Table 4.2, the generated predictive model based on the *Automatic feature selection* subset (using the F-statistic method) has a MAPE of 23.22% which is even worse than the case of *All predictors* where no feature reduction is applied. For the case of *All predictors*, a MAPE of 22.7% was achieved. This also shows that leveraging the automatic feature selection methods (e.g., F-statistic) does not improve the quality of results in this case. By generating a predictive model based on *only GPGPUs' architectural predictors*, it could improve the estimation quality to a MAPE of 21.60%. However, for both experiments, the R^2 has a negative score, which indicates that the regression model does not fit the data. In the case of generating the predictive model based on *Only PTX predictors*, it could obtain a better result in terms of RMSE and R^2. However, the MAPE of 24.12% indicates the highest error percentage overall in Experiments.

Based on our experiments, the best power consumption predictive model is obtained by combining the features described in Table 4.3. The power consumption

Table 4.2: Experimental results for different input feature combinations

Training dataset subsets	**k**	**RMSE**	**MAPE**	**R2**
Automatic feature selection	3	25.784	0.2322	0.3417
All predictors	18	33.877	0.2270	-0.1934
Only architectural predictors	1	33.2414	0.2169	-0.1613
Only PTX predictors	7	29.8188	0.2412	0.0654
Best predictors combination	5	13.657	0.08849	0.8156

Table 4.3: Descriptions of the best predictors (features) combination

Features	**Brief description**
CUDA Cores	Number of CUDA cores the GPGPU has
RAM	Amount of GPU memory
Base Frequency	The base frequency of the GPGPU cores
Storage Speed	Bandwidth speed for Storage access
GFLOPS	Number of Floating Point Operations per Second
Memory Clock	Frequency of GPGPU memory
L2 Cache size	Size of L2 Cache of GPGPU in KB
ret	Number of PTX instructions
trainable params	Number of learnable and changeable parameters

predictive model has an R^2 of 81.56%, indicating a high correlation between the chosen input features and power consumption. In this case, a MAPE of 8.8% is obtained.

To validate the quality of the generated predictive model, it is applied the proposed approach to various new CNNs that have yet to be used in the training phase. Fig. 4.3 illustrates the predicted (in blue) and the original value of power consumption (in orange) for 12 different CNNs on the NVIDIA GTX 1080Ti. As shown in this figure, the prediction is nearly identical to the original value for some CNNs, such as *Vgg19* and *Vgg16*.

Besides the K-NN predictive model, also a NN-based predictive model is implemented. The experimental results demonstrate that the power estimation based on the PTX and GPGPU architecture is also promising with NN. Fig. 4.4 illustrates the results of the NN-based predictive model. On average, our predictive model achieves an Absolute Error (AE) of 8.3%. The best-case result belongs to the power prediction for the ResNet152 with an AE of 0.73%. The worst-case

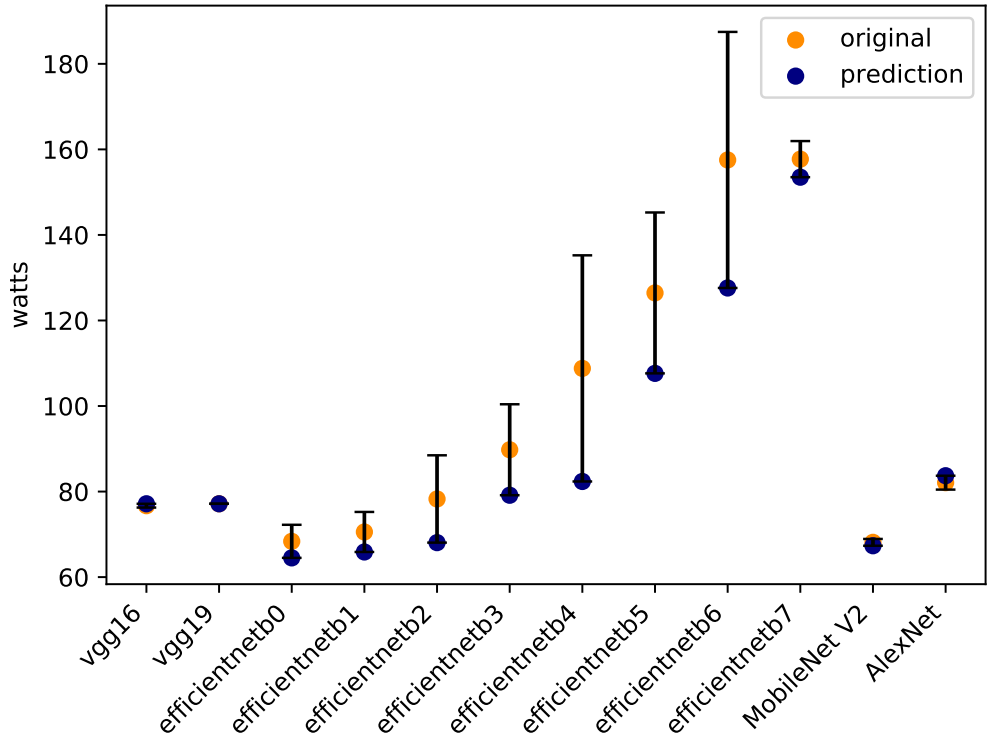


Figure 4.3: Scatter plot of the predicted power consumption for different CNNs on the NVIDIA GTX 1080 Ti with 8GB Memory.

forecast relates to DenseNet201 with an AE of 15.75%. This is because more ResNet variations are included in the training data than, e.g., Densenet.

4.4 Discussion

The results are discussed, and the research questions for this chapter will be answered in the following section.

Research Question 4.1 Which ML Model:, our experimental results illustrate that the best results are achieved with the K-NN, which performs slightly better than the NN-based predictive model. In comparison with an average MAPE of 8.4% for the power consumption for the NN-based predictive model, the K-NN-

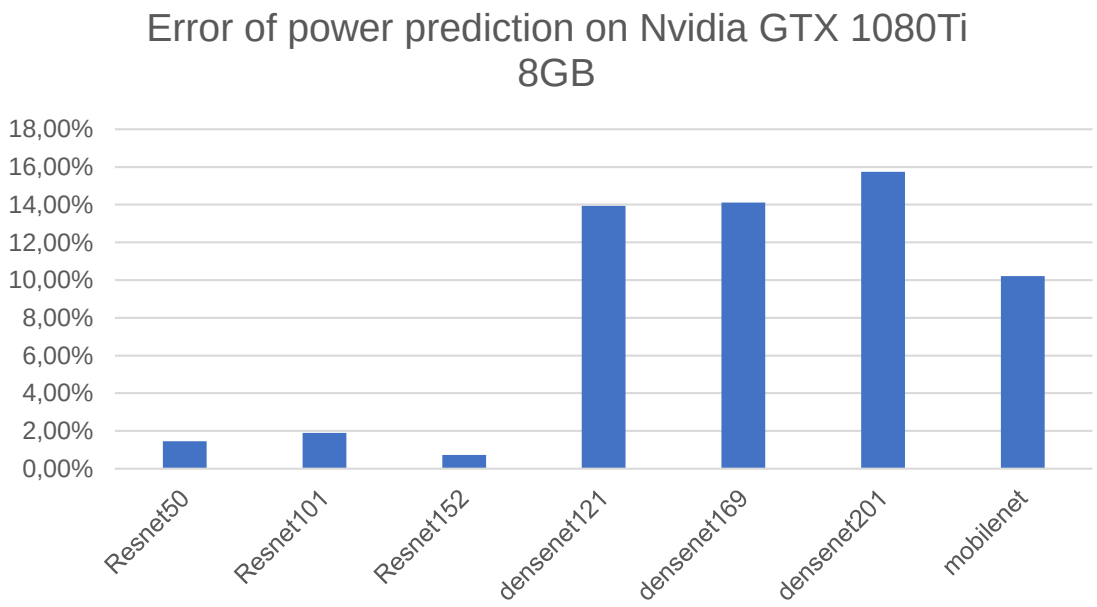


Figure 4.4: AE of power estimation of different CNNs for NVIDIA RTX 1080Ti.

based predictive model achieves 0.4 percentage point better accuracy on average. However, in the best case, the K-NN-based predictive model has better accuracy and a lower Absolute Percentage Error (APE). The best power consumption estimation result belongs to the *Vgg19* CNN with 0.0003% APE while the estimation result using the NN-based predictive model reports an APE of 0.73%. Another essential point that must be taken into account is that the K-NN-based predictive model only needs nine instead of 29 features used by NN-based predictive model to achieve these results [5, 6]. Consequently, interpreting the generated power consumption predictive model using the K-NN-based predictive model [6] is much easier than the NN-based predictive model [5]. Due to the fewer features, understanding the importance of features is more straightforward for designers, which can further help them design space exploration. Moreover, the experimental results demonstrate that the proposed approach based on the K-NN regression provides designers with an easy-to-interpret and fast power consumption estimation solution, obtaining promising results even with small training data.

However, the data set is relatively small for a NN-based predictive model [5].

This can change on a more extensive training set covering more GPGPUs and different types of NN and not only CNNs.

Research Question 4.2 Most impact features: the following features are having the most impact on the prediction regarding the best predictive model: CUDA cores, Random Access Memory (RAM), base frequency, storage speed, Giga Floating Point Operations per Second (GFLOPs), memory clock, L2 cache size, number of PTX instructions and trainable parameters.

Research Question 4.3 What k for K-NN as the experimental results show that in most cases, there is no significant improvement of K-NN results for a k larger than 5. The best results are achieved by K-NN with a k of 5.

Increasing the variation of CNNs can further improve the prediction. Moreover, larger training data sets can improve performance for NN. As K-NN is sensitive to the amount of data points in the training data set, the runtime will deteriorate. Consequently, K-NN should only be applied to small data sets. Hence, NN is a good option on large training data sets while K-NN is to be prioritized on smaller ones.

4.5 Conclusion

This chapter presents a novel power consumption estimation approach for CUDA-based CNNs on GPGPUs based on the nonparametric K-NN regression method compared to NN-based prediction. This chapter has illustrated how the power consumption of a given CNN on a GPGPU can be estimated by analyzing its PTX code, CNNs' topology, and GPGPUs' architectural Information. Moreover, it is been shown that promising results on even small training datasets can be achieved using the K-NN regression with beneficial advantages compared to NN-based approaches like a smaller number of input features and more minor AE in best-case prediction.

One of the primary usefulness of the proposed approach is for the DSE of IoT and Edge devices where CNN algorithms need to be implemented, urging the increasing use of hardware accelerators (e.g., GPGPUs). In this case, the proposed method can provide designers with early power consumption (one of the

crucial design constraints) estimation of a given CNN model on different GPGPUs at the time of compilation. Experimental results on various CNNs demonstrated the advantage of our approach in power consumption estimation.

5 Power and Performance Analysis with Dynamic Frequency Scaling

In this chapter, an analysis of the performance of CNN executed on GPGPU with DFS is presented. It was found that changing the frequency significantly impacted power consumption but only had a marginal effect on computation time. Furthermore, increasing the frequency beyond 1200 MHz no longer improves computation time. Therefore, a lower frequency can help create an energy-efficient CNN inference system without sacrificing performance. Additionally, a new approach that allows designers to estimate the power consumption of CNN's inferencing on GPGPU with DFS quickly and accurately during the early stages of CNN's development is presented. Multiple ML techniques are evaluated for the predictive model generation, and the best performing one is selected to implement in our novel approach. The proposed approach uses static analysis for feature extraction and Random Forest Tree (RFT) regression analysis for predictive model generation. Experimental results demonstrate that our approach can predict the CNNs power consumption with a MAPE of 5.03% compared to the actual hardware. The presented research is based on [7, 52]. Moreover, the methodology of [5, 6] is evolved for DFS.

The chapter is structured as follows: in Section 5.1, a brief introduction is given, followed by an introduction of the methodology in Section 5.2. The experimental results are presented in Section 5.3 and discussed in Section 5.4. The chapter closes in Section 5.5 with a conclusion.

5.1 Introduction

GPGPUs require high energy consumption. For example, to achieve high performance, the *Summit* supercomputer uses 27,648 NVIDIA Volta GPGPUs, leading to high energy consumption where a power supply of 13 million watts is required [19]. Data centers and cloud systems can offer a higher number of GPGPUs for HPC. However, due to power consumption limitations, only a limited number of GPGPUs with specific configurations, in terms of power consumption, are currently available for most emerging technologies applications. On the other hand, due to the emerging usage of ML algorithms in IoT devices embedded and edge computing systems, the need for HPC has been significantly increasing. This leads to increased power consumption of almost all machines using ML algorithms (e.g., CNNs). Hence, finding a trade-off between power consumption and HPC is paramount for such applications.

Performing DSE is essential to find such a trade-off, especially for applications with a limited power supply like IoT and edge devices with unlimited power supply. A promising power management technique that is widely used during DSE is Dynamic Voltage and Frequency Scaling (DVFS) or DFS. The DVFS technique changes the voltage/frequency during the processing of applications, while DFS only changes the frequency. Both energy consumption as well as performance can be optimized with these techniques. While the DVFS/DFS techniques for CPU-based applications are well-developed, in the case of the GPGPU, the study started only a few years ago [99]. Moreover, [100] pointed out that DVFS techniques for CPU do not suit GPGPUs. Consequently, new strategies must be developed for GPGPUs.

GPGPU simulators can be used for DSE to support system designers in finding suitable devices. Those simulators deliver details of GPGPU kernels to understand the execution behavior on GPGPUs [43]. Therefore, they use performance counters and specific hardware details to estimate the execution time and power consumption. They reach an accuracy between 10% to 20% compared to real hardware [39]. However, these simulators require a long execution time and are significantly slower than native execution on real hardware [39, 43].

To tackle this issue, several predictive models have been developed [5, 21,

99, 101]. They can be classified either as empirical or statistical studies. The first relies on code analysis, while the second relies on the program performance counter. Empirical approaches can be called bottom-up approaches and usually require detailed information about the GPGPU micro-architecture. The statistical approaches ignore the GPGPU architecture and treat it as a black box. These approaches take details of the application behavior and analyze the relationship between performance counters, GPGPU power consumption, and runtime [99].

Although the aforementioned methods can help designers build fewer prototypes during DSE and avoid costly design loops, they need detailed application behavior analysis and specific profiler and profiling settings to collect the necessary performance counter. Moreover, most do not support power prediction in the case of DFS.

This chapter presents a fast and accurate predictive model considering the DFS ability of GPGPUs and straightforward collectible predictors that do not need specific profiling settings. Compared to other empirical studies, there is no need for a detailed break-up of the GPGPU micro-architecture and treat most parts of the GPGPU architecture as a black box. Therefore, the method introduced in chapter 4 is evolved considering different frequency settings of GPGPUs. Thus, combining the advantages of both empirical and statistical approaches leads to a predictive model that achieves a MAPE of 5.03%.

In summary, the main contributions of the chapter regarding power consumption estimation are as follows:

1. A fast and accurate power estimation model considering DFS to perform Neural Hardware Search (NHS) for various configurations of GPGPUs,

2. evaluation of different ML techniques to obtain the best predictive model (i.e., Random Forest Tree regression analysis),

3. evaluating the applicability and accuracy of the proposed approach in estimating power consumption of 30 standard CNNs for the NVIDIA V100S GPGPU (which is one of the most used GPGPUs in data centers).

Furthermore, the performance analysis yielded the following results:

1. There is almost no correlation between performance and frequency on most CNN inferencing tasks,
2. there is a strong correlation between power consumption and frequency,
3. the power consumption is strongly increasing for frequencies larger than 1200 MHz,
4. lower frequencies do not lead to significant performance loss but reduce power consumption.

5.2 Methodology

The proposed methodology is structured into two main phases, which are 1) training dataset creation and 2) predictive model generation and evaluation. The overall flow of the proposed methodology is illustrated in Fig. 5.1. In the following, each phase of the proposed approach is explained in more detail.

5.2.1 Training Data Generation

An HPC Cluster with Simple Linux Ressource Manager (SLURM) is used for training data creation. The used machine is equipped with three NVIDIA V100S 32GB, 256GB memory, and 2 AMD EPYC ROME 7272. Since the system is a computing cluster, the Home directory is a *Network Attached Storage* (NAS)[1]; a 10GBit/s ethernet connection connects it.

To consider DFS, it is necessary to use the functionality supplied by the *nvidia-smi* tool to set a fixed execution frequency [102]. This makes it possible to execute the CNN benchmarks on frequencies between 1597 MHz and 135 MHz on the NVIDIA V100S GPGPU. Therefore, a SLURM-based HPC system application (e.i., a batch script) is developed to start a Job array for all available frequencies and CNNs combinations, hence, the benchmark can be run automatically. This

[1]Please note that the abbreviation NAS can stand for Neural Architecture Search as well as Network Attached Storage. In this thesis, the abbreviation NAS is used for Neural Architecture Search.

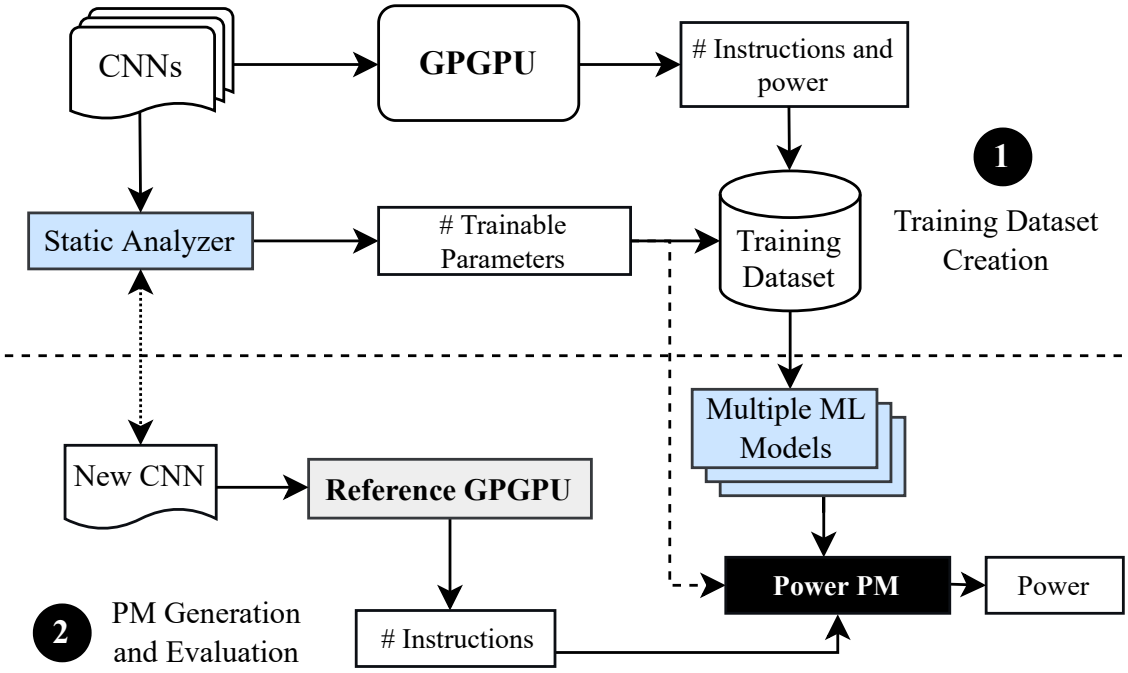


Figure 5.1: DFS-based power estimation methodology.

program is illustrated in Fig. 5.2. To avoid any side effects, it is been ensured that the benchmark is the only running job on the machine. This is done with the SLURM parameter *–exlusive* (Fig. 5.2, Line 5). The benchmark has three steps: 1) setting the frequency, 2) executing the CNN and measuring the power consumption and execution time, and 3) resetting the frequency for the subsequent execution (Lines 30 to 34). After the execution of CNN benchmarks on a given frequency, the measurements are added to a Comma-Separated Values (CSV) file containing the GPGPU name, CNN name, frequency, power consumption, and execution time. The observation is extended during further data collection steps for the predictors.

To get the exact number of executed instructions for CNN benchmarks, the NVIDIA Profiler *nvprof* is used, which analyzes the CNN during execution [103]. Thus, all 30 CNN benchmarks are executed and create a profile for each. The extracted number of executed instructions extends the training dataset. The main reason is that the total number of PTX instructions cannot be extracted from abstract PTX files at compilation time as it does not contain dynamic informa-

```
1  #!/bin/bash
2  #
3  #SBATCH --job-name="Power_Benchmark"
4  ...
5  #SBATCH --exclusive
6  ...
7  declare -a CLOCK_RATES
8  CLOCK_RATES=(1597 ... 135)
9
10 declare -a CNNs
11 CNNs=('m-r50x1' ... 'alexnet')
12
13 declare -a combinations
14 index=1
15
16 for clock_rate in ${CLOCK_RATES[*]}
17 do
18 for CNN in ${CNNs[*]}
19 do
20 combinations[$index]="$clock_rate $CNN"
21 index=$((index+1))
22 done
23 done
24
25 parameters=(${combinations[${SLURM_ARRAY_TASK_ID}]})
26
27 clock_rate=${parameters[0]}
28 cnn=${parameters[1]}
29
30 nvidia-smi -pm 1 -i 0
31 nvidia-smi -i 0 -ac 1107,${clock_rate}
32 srun python3 benchmark.py -n ${cnn} -f ${clock_rate}
33 nvidia-smi -rac
34 nvidia-smi -pm 0 -i 0
```

Figure 5.2: Part of SLURM SBATCH Job script to execute the CNN benchmarks with different frequencies on the experimental setup.

tion such as the length of loops or jump instructions based on the comparison (Fig. 2.3, Lines 14 and 15). Thus, at least one execution of the CNN benchmarks on real hardware or cycle-level simulators (which take much longer than real devices) is required. In the *Training Dataset Creation* phase, the NVIDIA V100S GPGPU is used to measure the number of executed instructions for each CNN.

To handle this issue for a new CNN in the second phase (Fig. 5.1, *Predictive Model Generation and Evaluation*), we take advantage of a reference GPGPU (which can be any available GPGPU) and the NVIDIA profiler *nvprof* to obtain the total number of instructions.

The trainable parameters are calculated by the (*Static Analyzer* module utilizing the TensorFlow functionality, see Fig. 5.1, Phase 1) for all 30 benchmarks

CNNs and add the total number of trainable parameters for each CNN to the corresponding observation of the DFS benchmark. The final training dataset D is defined as follows:

$$D = \{d_i | d_i = \{y_i(c_i, tp_i, ins_i)\}; 0 < i < n\} \tag{5.1}$$

For each observation d_i the parameter c_i, tp_i, ins_i identify the GPGPUs' frequency, the CNN trainable parameter and the total number of executed PTX instructions, respectively. The output parameter y_i denotes the measured power consumption (in watts) for each CNN running on GPGPUs. The training dataset is split into 70% training and 30% evaluation, which are independent and have no overlapping data. Afterward, the different predictive models (five different ML techniques are used) are trained on the dataset and evaluated in terms of accuracy and speed.

5.2.2 Differentiation of Methodologies

The methodology in this chapter differs from the methodology in Chapter 4 at two key points.

1) All available frequencies were included in creating the data set, whereas in Chapter 4, the GPGPU configuration was not manipulated, and the GPGPU, therefore, runs in the standard configuration. This results in significantly more data and measurement points per GPGPU than in the method from Chapter 4.

2) Instead of analyzing PTX code, the instructions are measured here by nvprof Profiler. This leads to significantly more accurate results than static code analysis. Using nvprof also has a disadvantage, as a one-time execution on a GPGPU is required. Moreover, using nvprof also results in a further difference, as the nvprof measures the SASS instructions, not the PTX.

5.2.3 Performance Evaluation Process

As already mentioned, the following setup is considered: the independent variables contain the CNNs used for inferencing $N_i \in \mathcal{N} = \{N_1 = alexnet, N_2 = densenet121, ..., N_{31} = xception\}$ and the varied frequencies $f \in \mathcal{F} \subset \mathbb{R}_{\geq 0}$. The

dependent variables contain the maximum power consumption P and the computation time T. While $\mathcal{N}$ is a discrete set, $\mathcal{F}$ can be easily discretized by sampling the frequencies at interest and ensuring a sensible distribution.

For each CNN inferencing task, 196 samples of different frequencies (ranging from 135 MHz to 1597 MHz) in approximately uniform distribution are collected and performed $n = 3$ repetitions of these tasks to factor out measurement noise. Hence, for each CNN, $196 \cdot 31 = 6076$ data points exists (each in $n = 3$ repetitions) both in maximum power consumption and computation time.

For the analysis standard statistical measures are used for evaluating the results: First, the reliability of the computed data is analyzed by computing mean $\mu(d)$ and standard deviation $\sigma(d)$ for each data point d in maximum power consumption P and computation time t, as well as the variation coefficient $\frac{\sigma(d)}{\mu(d)}$ for both dependent variables. Second, the relative values are computed (on means) $t_{rel}(d) = \frac{\mu_t(d)}{\min_{f \in \mathcal{F}} t(d)}$ for computation time based on the minimum value of computation time. This computation shows for each frequency in each CNN by what factor the computation time increases compared to the shortest computation time given this CNN. Third, the correlation coefficients are computed as follows: $\sigma_{xy} = \frac{\mu(xy - \mu(x)\mu(y)}{\sigma(x) \cdot \sigma(y)}$; for each CNN between both dependent variables, but also power consumption to frequency as well as computation time to frequency, i.e., σ_{Pt}, σ_{Pf}, and σ_{tf} to analyze their influence on one another. This value is normalized to lie between -1.0 and 1.0, where higher absolute values denote a higher linear dependence and lower absolute values (usually below 0.5) indicate no linear correlation.

5.2.4 Predictive Model Generation and Evaluation

The analysis considered five common ML techniques for regression analysis, namely 1) K-NN, 2) Decision Tree, 3) Random Forest Trees, 4) eXtreme Gradient Boosting (XGBoost), and 5) Linear Regression.

As the K-NN worked well in chapter 4 it is used again and compared to other techniques besides NN. Although it is designed for classification tasks, it can be used for regression analysis, too [96] (for more details, see Section 4.2.4). Decision Tree, Random Forest Tree, and XGBoost are tree-based decision algorithms

that are usually used for classification but can also be used for regression analysis. The Decision Tree technique builds a binary tree structure, where a predictor defines each node and threshold [104]. The Random Forest Tree is an extension of this simple Decision Tree technique and consists of a collection of Decision Trees that creates a "Forest." Hence, the Random Forest Tree usually has better results than a single Decision Tree [105]. The XGBoost is a technique to speed up the runtime of tree-based ML techniques. Consequently, the execution time of XGBoost models is faster than for other tree-based ML algorithms [106]. However, to find the technique that provides the best predictive model, five different models are trained and compared in terms of accuracy and speed.

To compare different ML techniques and evaluate the generated predictive models, the following error metrics are used: 1) the MAPE and 2) the R^2 coefficient. This ensures that the accuracy of all predictive models is calculated using the identical metric, providing a basis for comparison. The MAPE is calculated based on Eq. (5.2) and the R^2 based on Eq. (5.3).

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}|\frac{y_i - \hat{y}_i}{y_i}| \tag{5.2}$$

$$R^2 = \frac{\sum_{i=1}^{n}(\hat{y}_i - \overline{y})^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2} \tag{5.3}$$

Here, y_i, $\hat{y}_i$, and $\overline{y}_i$ identify the observation, prediction, and average output values, respectively. In addition, this also allows for comparing the results with the latest works in the field.

5.3 Experimental Results

The experimental results and evaluations are two-fold. In Section 5.3.2, the experimental results on modeling power consumption with ML-based techniques for different frequencies of the GPGPU are presented. Afterward, the impact of different frequencies on the performance (e.g., the execution time of the CNNs) is

analyzed in Section 5.3.1.

5.3.1 Performance Behavior Analysis

In this section, the results of the above-described evaluation process will be presented. Furthermore, the results are analyzed and discussed to answer the following evaluation questions:

Research Question 5.1. *How reliable is the technical setup?*

Research Question 5.2. *What influence does the frequency have on the computation time of the CNN inferencing task?*

Research Question 5.3. *What influence does the frequency have on the power consumption of the CNN inferencing task?*

Research Question 5.4. *Which influence is higher on computation time or power consumption, the CNN's or frequency's influence?*

Combining the results of these evaluation questions may allow a sound answer to the original question about the influence of frequency scaling on performance measures.

Statistical Evaluation

Figures 5.3 and 5.4 show average computation time and maximum power consumption results. In both figures, facets show the results per CNN, the x-axis shows different frequencies, and the y-axis shows the average value with an error band of one standard deviation for time and power, respectively.

In Figure 5.3, one can see that in most CNNs, the error band is comparatively narrow, with the apparent exception of the densenet-variants. The inceptionnet-variants and efficientnet-variants also show higher standard deviations of consumption time than the remaining CNNs. It is noticeable that the frequency seems to have, at most, a mild negative effect; for most CNNs, the trend appears constant at first sight.

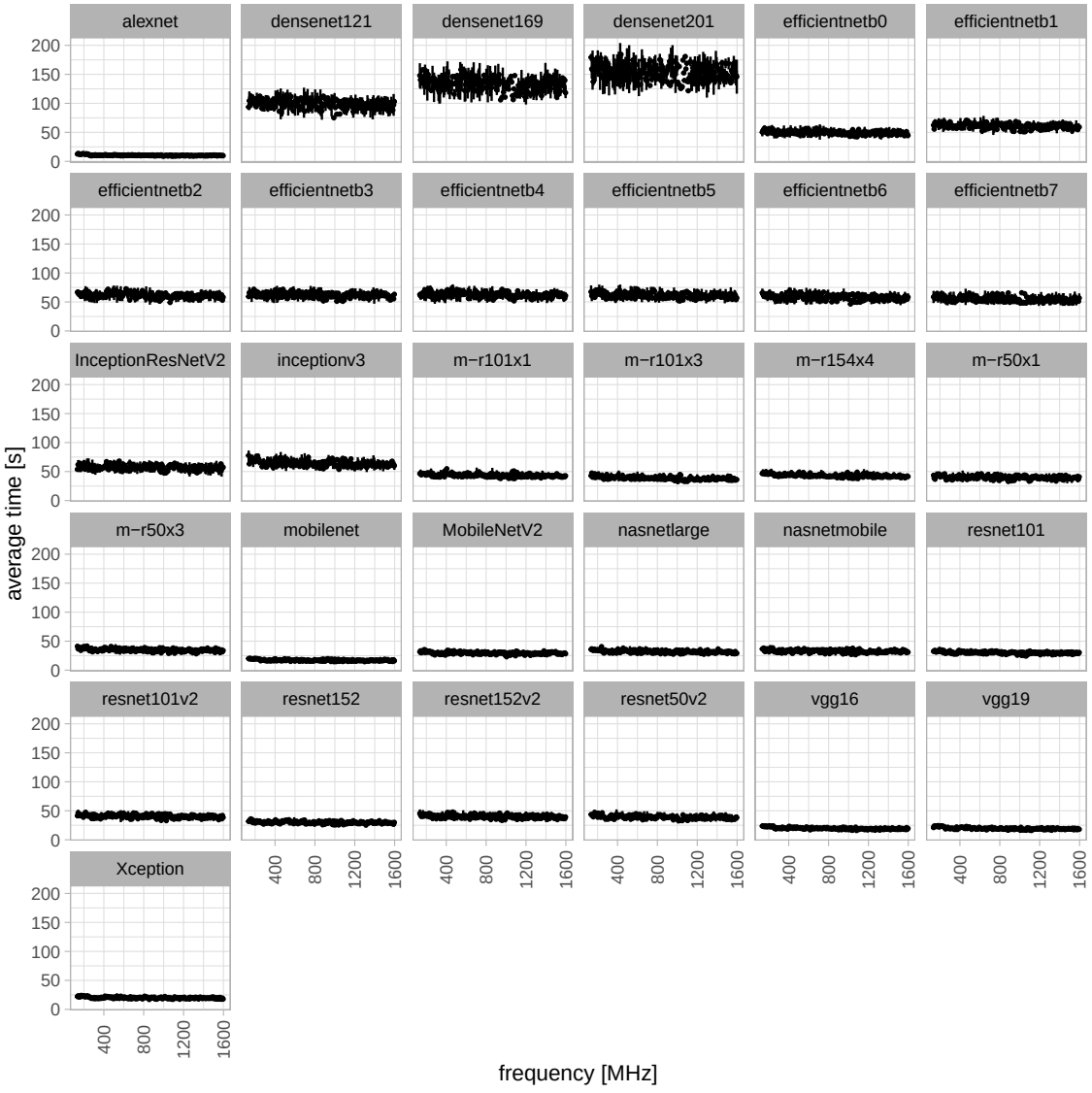


Figure 5.3: Average computation time (in seconds) for all CNNs and frequencies with error bands of one σ

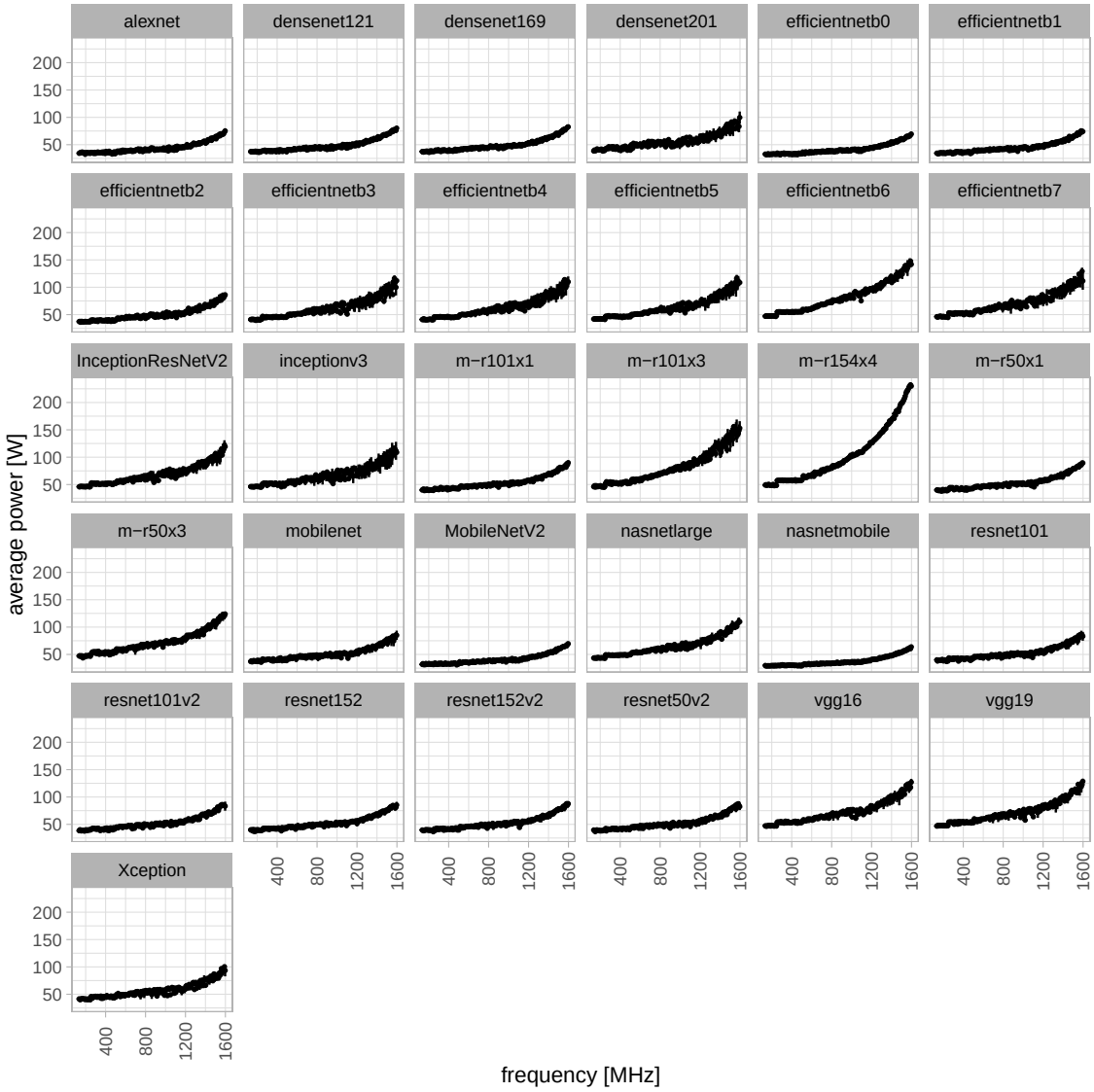


Figure 5.4: Average maximum power (in W) for all CNNs and frequencies with error bands of one σ

In Figure 5.4, illustrates again higher standard deviations for most of the efficientnet-variants and the inceptionnet-variants; however, not for the densenet-variants. Here, only densenet201 shows high standard deviations—additionally, the setup results in higher standard deviations for the m-r101x3. As for the average maximum power values, the figure shows a progressing curve for all CNNs. However, with varying steepness, e.g., the m-r154x4 has the highest increase for high frequencies. In contrast, Nasnetmobile has a comparably low rise, although it is still an increase. It is, however, remarkable that the growth significantly progresses after passing the 1200 MHz frequencies. This is where the most significant increase starts for most CNNs.

To further investigate the only barely visible negative trend for the average computation time, the relative average computation is described in Section 5.2. Figure 5.5 shows the result. Again, each facet represents a CNN inferencing task (in the same order as both figures before), the x-axis shows the frequencies, and the y-axis now shows the relative average computing time based on the minimum value (for this CNN). The blue lines indicate a model fitted via the loess method to aid the eye in visualizing the underlying trend. It cannot be seen as direct model computation. There are several things to observe here: First, the higher variability in the data is not an indicator of a high deviation in terms of the frequency variation as all of the data is now scaled to its perceptive minimum and does still not exceed 1.5, i.e., a 50% increase on the minimum. Second, in this visualization, the negative trend is more straightforward to spot. However, it is still minimal. Some CNNs, e.g., alexnet, vgg16, vgg19, show a relatively straightforward negative trend, while others still show none at all, e.g., efficientnetb3, densenet201, or InceptionResNetV2. Third, the minimum value, i.e., 1.0, generally lies between 900 and 1200 MHz, sometimes even once at the beginning and end of this range, e.g., InceptionResNetV2. Last but not least, some CNNs seem to show a change of trend in the middle of their data: It is especially prominent in densenet169 and nasnetmobile, where at around 900 MHz, there seems to be a drop in relative computing time, and then an increase can be seen.

The correlation coefficient for each CNN between frequency, average computation time, and average maximum power is computed to undermine our visual findings with a quantitative metric. Figure 5.6 shows the results; the light gray

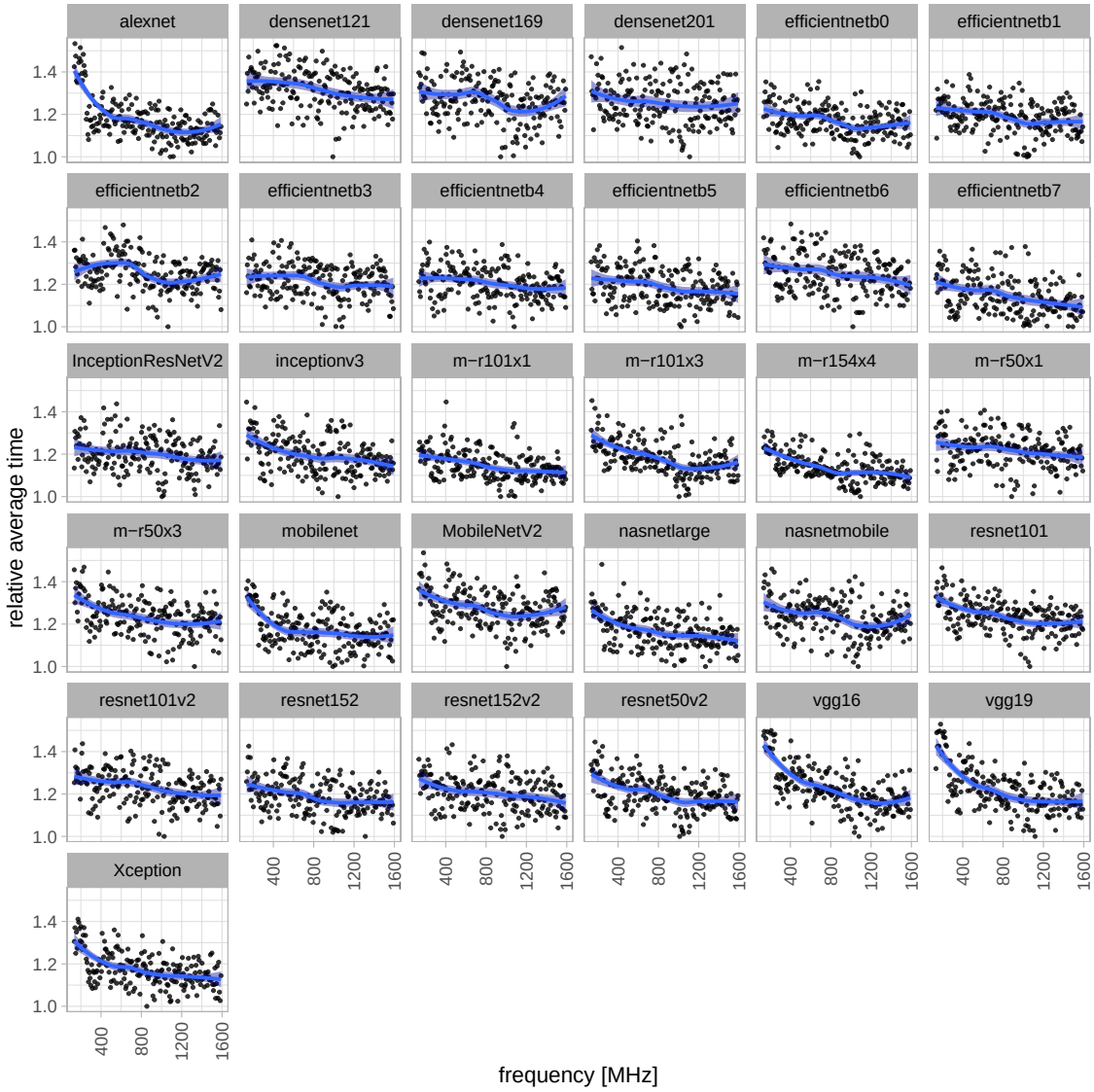


Figure 5.5: Relative average computation time for all CNNs and frequencies with a loess-model fitted

columns show the correlation coefficient results for frequency and average maximum power, and the dark gray columns show the correlation coefficient results for frequency and average computing time. As was already visible in Figure 5.4, the values for the correlation coefficient with average maximum power are very high, sometimes even close to 1.0, i.e., there is a strong positive relationship between both variables. To phrase it differently, a higher average maximum of power can be seen for higher frequencies. The picture is inverse when looking at average computing time. Here, all correlation coefficients are negative, albeit very small. The highest values to be observed are the ones for alexnet and both vgg16 and vgg19, which matches the conclusions from the last paragraph.

5.3.2 Power consumption Modeling

The results of the predictive model generation are presented below. The evaluation of the created predictive models aims to answer the following research questions discussed in sec 5.4.

Research Question 5.5. *Which ML technique performs best for power estimation of CNNs inference on GPGPU?*

Research Question 5.6. *Can the use of predictive models reduce the time required for DSE?*

The initial experimental results sound promising. Five commonly used ML algorithms are evaluated as predictive models for power consumption considering DFS. The experimental results are consolidated in Table 5.1, which shows the accuracy of the predictive models for average, top 10%, and bottom 10% prediction. The Linear Regression shows the worst result with a MAPE of 15.31%, followed by the K-NN with a MAPE of 7.74%. The tree-based ML algorithms, Decision Tree, XGBoost, and Random Forest Tree, attained better results with MAPE of 6.03%, 5.43%, and 5.03%, respectively. The best predictive model is based on the Random Forest Tree algorithm, where the number of instructions and the fixed frequency are used as predictors. In this case, the Random Forest Tree predictive model achieves a MAPE of 0.3% for the top 10% and 5.56% for the bottom 10% prediction. Since the features are simple to collect and consist

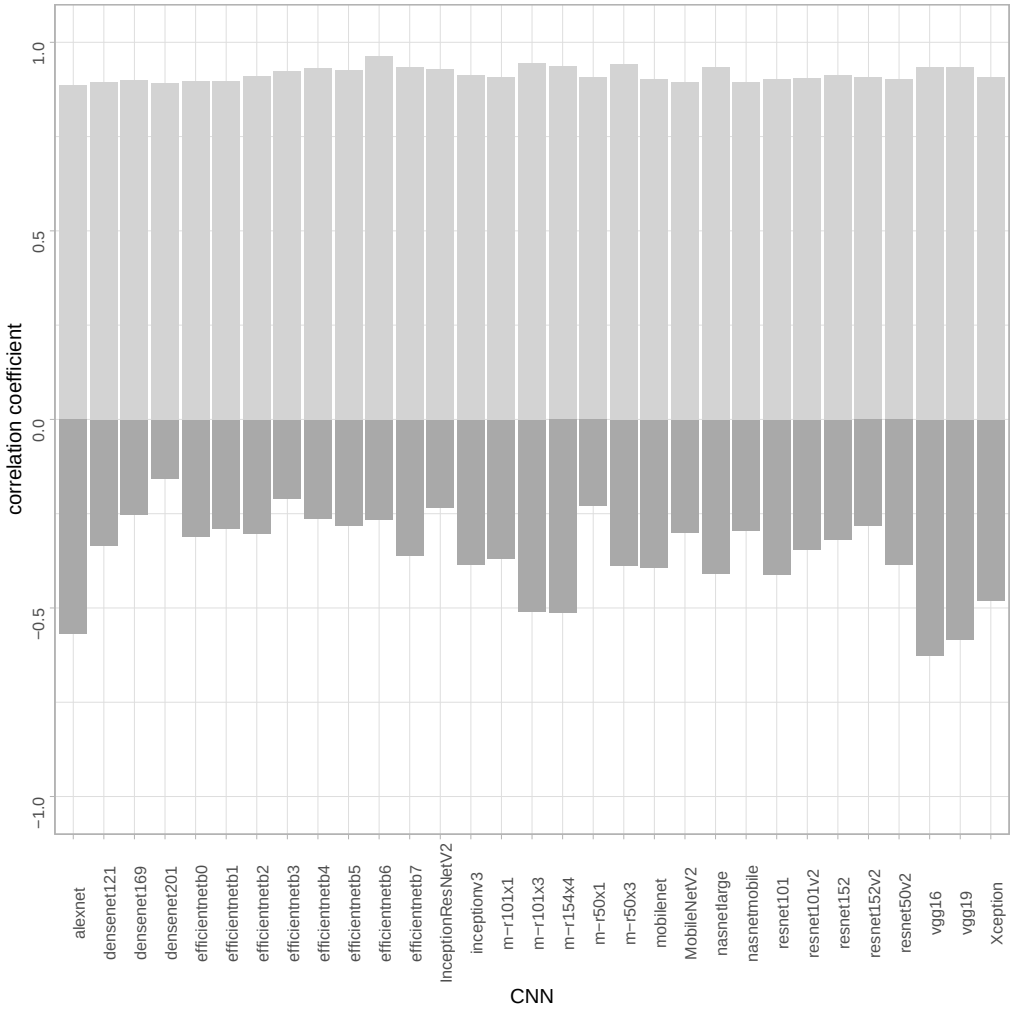


Figure 5.6: Correlation coefficients for frequency and computation time as well as frequency and power

of only two elements, the proposed approach can be easily applied. This can significantly enhance the DSE process, avoid the heavy tasks of specific setup and configuration, and extract many features usually required by most other approaches.

Compared to the state-of-the-art approach [6] with a MAPE of 8.3%, our proposed approach achieves $1.65\times$ better accuracy. The main reason is that the proposed approach is specified to a single GPGPU model and can predict the power consumption for various CNNs for this specific GPGPU. Please note that

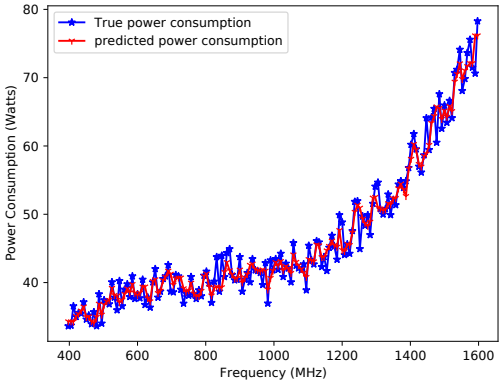
Table 5.1: Comparison of four different ML-regression algorithms in terms of accuracy and execution time

GPGPU	Regression Model	Accuracy					Execution Time	
		Average		Max Absolute Error	Top 10%	Bottom 10%	fast	slow
		MAPE	R^2		MAPE	MAPE		
V100S	Linear Regression	15.31%	0.6447	117.05 Watts	1.06%	16.88%	$3.5099e^{-5}$	0.0002
	K-Nearest Neighbors	07.74%	0.8027	93.09 Watts	0.52%	8.53%	0.0003	0.0017
	Random Forest Tree	**05.03%**	**0.9561**	**38.24 Watts**	**0.30%**	**5.56%**	**0.0050**	**0.0109**
	Decision Tree	06.03%	0.9359	38.38 Watts	0.32%	6.65%	$4.219e^{-5}$	0.0002
	XG Boost	05.43%	0.9512	48.13 Watts	0.34%	5.99%	$9.0800e^{-5}$	0.0003

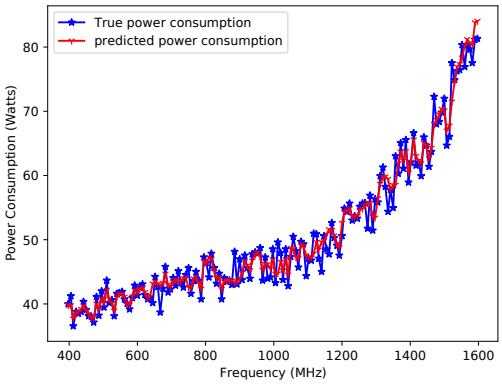
the GPGPU can be changed, and any other GPGPUs can be added to the training dataset. In this case, the predictive model can be adopted based on which GPGPU is considered the target device. So, any extension on the training dataset can be quickly set up.

The execution time of our approach is 0.0109 seconds in the worst case. Based on this, DSE for 30 different CNNs, and 196 possible frequencies of the NVIDIA V100S take about $30 \cdot 196 \cdot 0.0109s \approx 64.1s$ which leads to a tremendous speed-up of $11009\times$ in comparison to the naive approach where each CNN is executed for each frequency on an actual device (e.g., 195 hours on the NVIDIA V100S for 30 CNNs and 196 frequencies). Moreover, it opens another possible use case: online dynamic frequency scaling of GPGPUs. The predictive model can estimate the power consumption of a CNN online on the device and, thus, scale the frequency depending on the executed CNN in production. This opens additional power-saving options for systems executing different CNNs on the same device (e.g., GPGPU) like HPC-Systems or cloud providers. For example, the NVIDIA V100S has 196 frequencies between 1597 MHz and 135 MHz that can be configured. Consequently, finding the frequency with the lowest power consumption requires 196 execution of the predictive model takes $196 \cdot 0.0109s = 2.13s$. In the case of the model's periodic executions, the best frequency can be calculated and cached after the first inferencing and used in an additional execution.

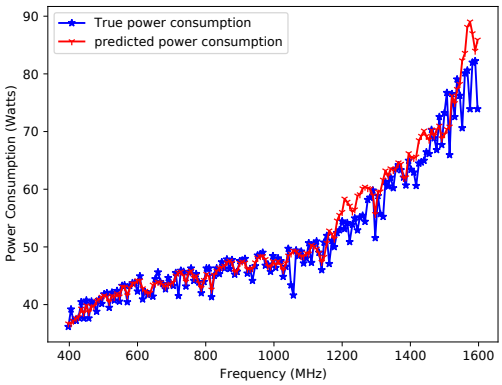
Based on our predictive model, a comparison between the predicted power consumption and the actual power consumption on a real device (e.g., NVIDIA V100S) is illustrated in Fig. 5.7. This figure shows that the predictive model is close to power consumption.



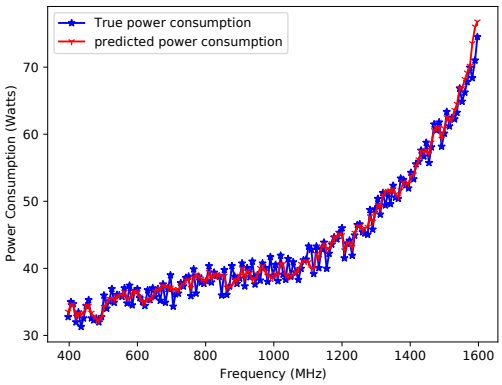
(a) AlexNet



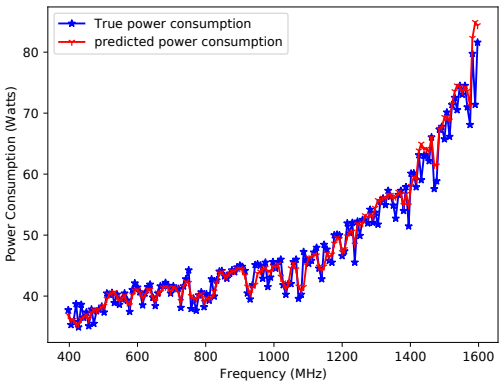
(b) DenseNet121



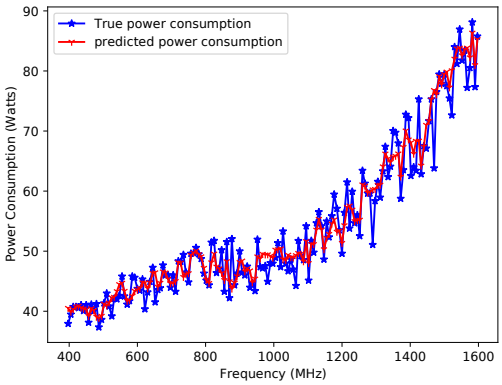
(c) DenseNet169



(d) EfficientNetB0



(e) EfficientNetB1



(f) EfficientNetB2

Figure 5.7: Comparison of predicted and real power consumption for six different CNNs for frequencies between 397 MHz and 1590 MHz on the NVIDIA V100S GPGPU.

5.4 Discussion

The following section discusses the results of the aforementioned analysis and the predictive model. Moreover, the results are used to answer the research questions of this chapter.

5.4.1 Performance Evaluation Discussion

Research Question 5.1 technical setup:, the repetitions must yield comparable results. A closer look at Figures 5.3 and 5.4 shows that although some CNNs have a higher standard deviation, these are also the ones with higher average values. Computation of the variance coefficient for the maximum power and computation time yields most variance coefficients smaller than 0.2 and none exceeding 0.3. These small values indicate that it is valid to trust the technical setup to give reproducible values.
Research Question 5.2 influence on computation time, only a marginal negative impact can be detected on the computation time of the CNN inferencing task, i.e., higher frequencies lead only to tiny improvements in computation time, if at all. This finding aligns with the general direction of the conclusions of [107].
Research Question 5.3 influence on power consumption:, a significant link between frequency and power usage can be observed. The power consumption drastically increases when the frequency exceeds 1200 MHz, the range where overclocking begins, as the base frequency of NVIDIA V100 is 1245 MHz. As a result, overclocking significantly impacts power consumption, while underclocking has a more negligible effect on most CNNs. However, some exceptions exist, such as the NASNetlarge or the NASNetMobile, which suggests that network design also plays a role in power consumption. Overclocking the GPGPU core frequency generally does not result in significantly better performance but increases power consumption dramatically.
Research Question 5.4 what does the frequency influence more: has two parts: Firstly, the frequency significantly impacts power consumption, as shown by the strong correlations. There are a few exceptions, like the NASNetLarge and NASNetMobile, where the increase in power consumption already starts before

the 1200 MHz frequency. Please notice that both CNNs are automatically designed by NAS techniques and not by humans. Thus, the NAS technique and the implementation of both networks could lead to different behavior. Secondly, while the correlation between frequency and power consumption is generally high, the correlation between computation time and frequency is the opposite, indicating that the application influences computation time more than the frequency. This effect can also be observed in the findings of [107] for other High-Performance Applications. In addition, the implementation of the application can also have a significant impact on the computation time.

Overall, lower frequencies lead to lower power consumption for CNNs, while the computation time is always at most 50% slower as the fastest execution. Consequently, it can be recommended to lower the frequency of CNNs inferencing tasks to establish sustainable and energy-efficient systems.

5.4.2 Predictive Model Discussion

A modification of the original power prediction approach from chapter 4 is presented. It illustrates how slight changes can enhance the method for DFS on GPGPU.

Research Question 5.5 best ML Model:, the Random Forest Tree performs the best for power consumption estimation of CNN inference on the NVIDIA V100S. However, the approach still has some drawbacks as it is limited to a single GPGPU and to cover multiple GPGPU models, a training data set for each model has to be created as well as the particular predictive model must be trained on the different training data sets.

Research Question 5.6 time reduction:, using predictive models based on machine learning techniques can greatly improve the time for DSE compared to naive approaches such as executing the CNN for all available frequencies and profiling the power consumption. Experimental results have shown that the predictive model outperformed the naive approach by a factor of 11009 (195 hours for the naive approach versus 64 seconds for the predictive model).

5.5 Conclusion

In this chapter, a novel ML-based approach to estimate the power consumption of CNNs for GPGPUs with DFS is presented, as well as an in-depth analysis of DFS impact on the performance.

It is illustrated how the power consumption of CNNs for a given GPGPU with various frequencies can be estimated by only considering two main features as predictors: the total number of executed instructions and trainable parameters (related to the CNNs topology). The proposed approach empowers designers to apply NHS and hardware-aware NAS, considering the power consumption of CNNs for GPGPUs. The model can predict the power consumption of various GPGPU frequency configurations without retraining the model. Experimental results sound promising, and this new line of research helps make power estimation CNNs for GPGPUs with DFS indeed a cross-cutting activity in the early stages of the design process.

6 Hybrid PTX Analysis

This chapter introduces HyPA, a hybrid PTX Analyzer that inspects PTX code statically and dynamically. HyPA implements a partly functional emulator that executes instructions that rely on runtime dependencies to count the number of executed PTX instructions and divergent branches. HyPA executes compiled kernels—the programs that run on GPUs—generated by the CUDA compiler and supports the full PTX 7.7 specification. Compared to standard profilers, the functional emulator allows significantly faster analysis of PTX code. The evaluation quantifies the increase in performance through benchmark runs. HyPA achieved speedups of up to 536% compared to the nvprof profiler. Moreover, HyPA can gather performance metrics beyond static analysis (e.g., branch efficiency) by a faster execution time than by profiling the application on an actual device. Finally, HyPA is provided as an open-source project[1] to help developers and system designers in further research and development. The presented research is based on a previously published conference paper [8].

The chapter is structured as follows: after a short introduction and motivation in Section 6.1, the methodology is presented in Section 6.2. Next, the results and discussion are illustrated in Section 6.3 and Section 6.4, respectively. The chapter closes with the conclusion in Section 6.5.

6.1 Introduction

Three strategies can achieve energy-efficient applications: 1) designing more energy-efficient devices, 2) optimizing the applications' implementation, or 3) optimizing the chosen device for the chosen application. Options 1 and 2 are usu-

[1] `https://github.com/chmetz/hypa`

ally long-term approaches, as pursuing the second strategy requires deep inside knowledge of the code and behavior of an actual device to maximize its utilization and regarding option 1, utilizing a GPGPU energy-efficiently is a complex and challenging task [108, 109]. However, the third option may allow users to supply only the performance necessary for their task and, thus, optimize energy efficiency. To do so, one needs information about the potentially employed devices, e.g., the different GPGPUs, for his specific task, e.g., a CNN-inferencing task. Standard metrics that are used for the prediction of power consumption and performance are the *branch-efficiency*, the *#instructions*, and the *#floating-point-operations* [5, 6, 7, 21]. Code profiling and execution analysis must be performed for all possible setups to gather these metrics. Academia and industry proposed different approaches for code profiling and analysis. The three main approaches—which we will describe in the following—are 1) classical profiler tools that measure performance counters during the execution of the application on an actual device, 2) simulators that simulate or emulate the actual device, and 3) static code analysis where performance and power prediction is calculated based on the application's source code. However, all approaches have their limitations, discussed in the following:

(1) Profiling tools like nvprof [103], CUPTI [110], or nsight[2] log different performance counters (e.g., the total number of executed instructions) during execution on an actual GPGPU [70, 78, 103, 111]. To find the optimal GPGPU, it's necessary to run the profiling process on multiple GPGPUs since the profiling results are restricted to the GPGPU in use. This leads to three main disadvantages of profilers.

- Performance counters are inconsistent across different GPGPUs. It means that the same performance counter can be calculated differently on different GPGPUs, or the corresponding counter may not exist. Hence, comparing these metrics can be challenging or even impossible.

- The profiling step is very time-consuming as its duration is significantly longer than the application run-time due to its dependence on the GPGPU

[2] `https://developer.nvidia.com/nsight-systems`

instead of the application [72, 78, 111]. Hence, profiling GPGPUs is not a time-efficient way of determining the most appropriate GPU.

- For profiling, access to the actual GPGPU is necessary. As results are not transferrable between different GPGPUs due to the dependence on their machine language [78], the user would need access to every GPGPU that is evaluated, which is very costly.

(2) Simulation techniques solve the availability problem of GPGPUs. They allow metrics computation without access to the GPGPU. In the past, tools like GPGPUSim [112] and Ocelot [113] were proposed. However, since they emulate GPGPU behavior and execute the application on CPUs, which do not offer the same high parallelism capabilities, the execution time takes much longer than the profiling on actual GPGPUs. Additionally, they do not achieve identical results compared to profiling. Hence, while helping to reduce costs, simulation techniques are less favorable in terms of time than profiling techniques.
(3) Another possibility is static code analysis. Its main drawback is the fact that it can not analyze dynamic behavior. Conditional jumps may have different control-flow paths in different threads, which can only be determined with a dynamic analysis. Therefore, the result of static analysis either underestimates (in the case of loops) or overestimates (in the case of non-entered ELSE branches) several metrics (e.g., numbers of instructions). Other metrics, like branch efficiency, are inherently based on dynamic analysis and can not be computed. Its advantage, however, is the small time consumption [35, 72, 79, 80].

The three existing methods have crucial drawbacks for gathering standard metrics to predict power and performance based on ML. Therefore, we propose an approach for computing the respective metrics, which combines the time-related advantage of static analysis with the advantage of simulations without having its time-related drawback. Our approach is based on the idea that it is not necessary to simulate the entire application on the GPGPU, but only those parts that influence conditional jumps and may, therefore, obfuscate the results of static analysis. The resulting hypothesis is twofold: First, a hybrid analysis approach is more time-efficient than the standard simulation approach, and second, it is not only more precise than a static analysis but is also capable of determining metrics

that are inherently based on dynamic analysis.

Thus, HyPA is created to consider dynamic dependencies within the code and can be used without executing the entire application (e.g., CNNs). The basic idea of our tool is to read all instructions, consider the number of threads, and search for dynamic dependencies. Afterward, a functional simulator executes only those PTX instructions on a CPU that rely on dynamic run-time dependencies for conditional jumps. By this, HyPA is overcoming the lack of speed of existing GPGPU simulators. The generated profiles can be used for power and performance prediction of GPGPU applications (e.g., [101]) or for better code understanding (e.g., [72, 78, 112]). HyPA gives detailed information on the number and type of instructions, floating point operations, and divergent branches.

Our main contributions can be summarized as follows:

1. A hybrid approach of PTX emulation to profile CUDA applications on a low-level code basis
2. An automatic extraction of the following PTX code metrics: number of instructions, floating point operations, number of divergent branches, and branch efficiencys
3. An implementation of HyPA as an opensource project to help developers, computer architects, and researchers in their work

6.2 Methodology

The PTX Code analysis is split into three parts 1) static code analysis and dependency detection, 2) dynamic code analysis and emulation, and 3) profile generation. The general workflow of HyPA is illustrated in Figure 6.1.

6.2.1 Running Example

A running example is used to explain the different steps of HyPA, which is shown in Fig. 6.2. Even if the function does not implement a useful algorithm, it can be used to illustrate HyPA. The code defines a function named `example` that

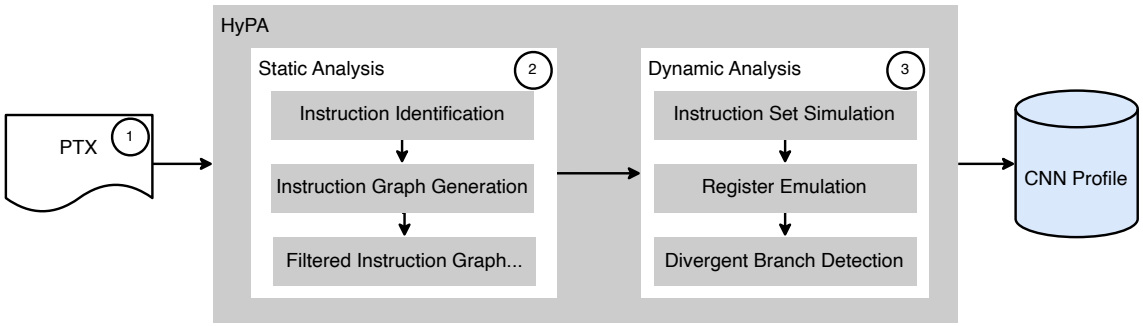


Figure 6.1: General workflow of the PTX analyzer HyPA

expects one parameter. The directive `.reqid` (see line 4) specifies how many threads should be used in the thread block. Lines 6 and 7 declare the number and type of used registers. In this case, there are two registers for predicates (boolean values) and four 32-bit wide unsigned integers. The remainder of the function consists mainly of PTX assembler instructions.

Following the structure is explained on the instruction `mov.u32 %ru1,%tid.x` from line 9. An instruction first states the instruction type. In this case, it is a `move` instruction that moves a 32-bit wide unsigned integer. After the instruction, the target register is specified, which is `%ru1` in this case. Please note that the maximum number of specified source registers is—according to the PTX documentation [76]—limited to four. Here, the instruction reads the register `%tid.x`. This is a special register that holds the identifier of the current thread and is set by the GPU. The `setp` instruction, for instance, used in line 11, compares two registers based on the operator, `<` in this case, and stores the result in the target register. Line 14 shows an example of a branch instruction `@%p1 bra then;` that jumps to the specified block `then` when the condition `%p1` holds. If the condition does not hold, the control flow continues with the next instruction.

6.2.2 Static PTX Analysis

HyPA starts with a static code analysis to detect all instructions that need to be executed, the number of threads that will be raised, and the run-time dependencies that need simulating to resolve conditional jumps based on dynamic dependencies. Therefore, HyPA parses the PTX Assembler and stores the information in

```
1  .visible .entry example(
2  .param .u32 param_0
3  )
4  .reqntid 256, 1, 1
5  {
6    .reg .pred    %p<3>;
7    .reg .u32     %ru<4>;
8
9    mov.u32       %ru1, %tid.x;
10   ld.param.u32  %ru4, param_0;
11   setp.lt.u32   %p1, %ru4, 1024;
12
13   setp.lt.u32   %p2, %ru1 128;
14   @%p1 bra      then;
15
16   mov.u32       %ru3, 32;
17   bra.uni       exit1;
18
19 then:
20   mov.u32       %ru3, 64;
21
22 final:
23   setp.gt.u32   %p3, %ru3 100;
24   @%p2 bra      exit1;
25
26   add.u32       %ru4, %ru4, 1024;
27
28 exit1:
29   ret;
30 }
```

Figure 6.2: Running example of PTX to illustrate the workflow of HyPA

an intermediate representation. In this intermediate representation, an instruction is formally denoted by a vector

$$ins = (id, dr, sr_1, ..., sr_m, CTAid, CBid) \tag{6.1}$$

where id stands for the unique instruction identifier, dr and $sr_j, 1 \leq j \leq m$ denote the destination and source registers, resp., and $CTAid, CBid$ are the CTA-Id and the Code-Block-Id. INS is the set of all possible instructions that enable the denotation of $\mathcal{I}(P) \in INS^n$ for the list of all instruction lines of a given program p. $\mathcal{I}_i, 1 \leq i \leq n$ denotes the i-th instruction line in this denotation scheme. Thus, the formal notation of the exemplary PTX code line `add.s64 %rd48 %rd7, %rd47;` is $ins = (0, rd48, rd7, rd47, 0, 0)$. Since it is the first identified PTX line, the ID id is set to zero (0). Moreover, since no CTA and labels are specified in the example, both CTA-Id and Code-Block-Id also are

set to zero (0) in this case. Please note, that the maximum number of specified source registers is limited to 4 based on the PTX documentation [76].

In the following, the definition of dependency graphs [114] and their restriction to conditional instructions are explained.

Definition 1 (Dependency Graph)**.** *A Dependency Graph G is defined as a tuple $G = (V, E)$, where V is a finite set of nodes, denoting instructions, and $E = \{(v_1, v_2)|v_1, v_2 \in V\}$ is a set of directed edges. An edge from node v_1 to node v_2 indicates that the latter instruction is dependent on the former.*

Each PTX code line containing an instruction ($\mathcal{I}_i$) is represented by a node ($v_i \in V$) where all instructions included in a CTA lead to a Dependency Graph G_{CTAid} containing these instructions as well as their data dependencies. A dependency of two or more instructions exists if a source register (sr_j) of instruction $mathcalI_i$ is the destination register (dr) of earlier occurring instructions. Consequently, the node v_i for instruction $\mathcal{I}_i$ depends on the nodes v_l whose instructions $\mathcal{I}_l$ have previously been written to the source registers specified for $\mathcal{I}_i$ and thus an edge $e = (v_l, v_i)$ is included in the graph for each instruction that does so [115].
During Dependency Graph (G) construction all jump instructions are stored in a Control Flow Instruction List (CFIL). At the same time, these nodes constitute the set $V_{branch} \subseteq V$, namely all nodes in the instruction graph that may affect a specific jumping operation. The jump instructions can be identified unambiguously based on the identifiers (id) assigned to the lines during initial parsing,

Filtered Dependency Graph

Based on our initial hypothesis, not all instructions must be executed to decide whether a thread will branch. Therefore, a dynamic slicing [115] is performed. Starting from the jump condition of a specific code block, all influencing nodes are traced back through the instruction graph. The resulting subgraph is called Filtered Dependency Graph (FDG) in the following. The FDG is formally defined as follows:

Definition 2 (Filtered Dependency Graph FDG)**.** *The Filtered Dependency Graph is*

defined as subgraph $G_{v^*} = (V', E')$ *of* $G = (V, E)$*, where* $v^* \in V_{branch} \subseteq V$*, with*

$$\begin{aligned} V' = \{v_0 \in V | \exists \pi = v_0 v_1 ... v_n \; with \; v_n = v^*, \\ (v_{i-1}, v_i) \in E, \; \forall i \in \{1, ..n\}\} \subseteq V \\ E' = \{e = (v_1, v_2) \in E | v_1, v_2 \in V'\} \end{aligned}$$

The FDG only consists of the instructions that need to be executed to identify if a conditional jump operation will be triggered or not. Hence, all jump operations have to be located to generate the FDG. Based on the CFIL from the dependency graph generation, the paths to the root nodes are identified. The working principle is as follows: A jump instruction from the CFIL marks the starting point at node (A) in the dependency graph (G). This starting node (A) is added to the FDG. For every added node, its (transitive) parents are added as well. The resulting graph trivially is a subset of G. The identified FDG will then be simulated during the Dynamic PTX Analyzation.

6.2.3 Dynamic PTX Analyzation

Based on the FDGs which are generated for all CTAs, the dynamic PTX analysis proceeds by emulating the GPGPU. The emulation does not perform the full application (i.e., PTX code) since the FDG only consists of a subset of instructions. Every generated FDG is given to the Instruction Set Simulator (ISS) which builds the core of the dynamic PTX analysis.

Instruction Set Simulator

Based on a C++ CPU implementation, the ISS executes PTX instructions included in the FDG. The necessary registers are emulated based on a symbol table, where each register is identified by its name and receives the belonging value after the instruction emulation. Moreover, the ISS receives the instructions, and based on a fixed assignment, the equivalent C++ implementation is performed. Nearly all existing PTX instructions are reimplemented in C++, allowing the analyzer to apply to all CUDA applications.

Register Emulator

To correctly determine the jump conditions, it is necessary to save the values of the registers. Hence, a data structure is designed that consists of a key-value pair that can be defined as a function based on the definition in Eq. 6.2.

$$f : K \rightarrow V \tag{6.2}$$

The required keys are determined based on the filtered instruction graph, and each key occurs precisely once. If the ISS now calculates a value, it is assigned as a value to the corresponding register key. This process is defined as follows:

$$f(k_i) \leftarrow v_{new} \tag{6.3}$$

The currently assigned value v_i is overwritten with the new value v_{new} if a register k_i is written to several times during the emulation.

In addition, the ISS reads the values of a register specified as source register for instructions from this data structure $f(k_i) = v_i$. This guarantees that during the emulation of the PTX code, the correct values are always present in the register emulator.

Divergent Branch detection

Each time a jump is performed, the current code-block ID is stored in a list. Each thread has its list of code-block IDs. The order of the IDs in the code-block ID list indicates the exact program path of the respective thread.

If the lists of all threads of a CTA are compared to each other, divergent program paths (e.g., divergent branches) can be recognized, and threads, which deviate, can be identified. When the code-block ID list is created, the first code block is always given the ID zero (0). Consequently, all code block ID lists start with ID zero (0), followed by the respective thread's code-block- ID sequence. Different orders of code-block IDs identify different code paths and thus divergent branches. If no

divergent branch occurs, all code path lists have the same order of code-block IDs.

Applications' Metrics

HyPA calculates metrics to generate an analysis profile for an analyzed PTX file. In the following, we describe the metrics calculated by HyPA and in which step they can be calculated. Therefore, we use the numbers 1 to 3 shown in Figure 6.1. The number ① means that the calculation is possible by simply scanning the source files. Number ② means that a metric can be calculated after the static analysis step of HyPA. Finally, the number ③ means that it is necessary to simulate the PTX program to gather the metric.

Number of CTAs ① The number of CTAs is an indicator for the number of existing different thread arrays. The higher the number is, the more different tasks are parallelized.
Calulation: Using the PTX code, HyPA extracts the number of CTAs.
Example: For the running example the *Number of CTAs* is 1.

Number of Threads ① The number of threads describes how many threads are executed, which can be used to calculate other metrics, such as *static instruction count*.
Calculation: The `.reqntid` directive is specifying the number of threads.
Example: In the running example, the *Number of Threads* is set to 256.

Number of Instructions ① As an indicator for the size of the analyzed PTX code, HyPA counts the number of instructions in the PTX code statically.
Calculation: During parsing, HyPA determines for each line if it is an executable instruction and sums up the number of executable instructions. However, as [35, 72, 79, 80] spotted, this leads to over or under-estimation as loops, and the application control flow is ignored.
Example: The number of *static instructions* for the running example is 12.

McCabe Complextiy ② McCabe Complexity is a measure for estimating the complexity of a function and is defined, according to literature, as:

$$McCabe = Number\ of\ binary\ branches + 1 \tag{6.4}$$

Calculation: HyPA uses its intermediate representation to count the number of binary branches.
Example: The *McCabe Complexity* for the running example is 3.

Statically Estimated Number of Executed Instructions ① The statically estimated number of executed instructions is the static approximation of the instructions to be executed at runtime based on *Number of Threads* and *Number of Instructions*.
Calculation: This metric is calculated as:

$$Number\ of\ Threads \cdot Number\ of\ Instructions \tag{6.5}$$

Example: The *Statically Estimated Number of Executed Instructions* for the running example is 3072.

Fraction of Executed Instructions ③ The ratio of executed trace instruction count to trace instruction count measures the performance improvement in the simulation due to the use of the filtered dependency graph. A value of 0 means that no instructions had to be simulated, while a value of 1 means that all instructions had to be simulated. *Calculation:* HyPA calculates the formula:

$$\frac{Executed\ Trace\ Instruction\ Count}{Trace\ Instruction\ Count} \tag{6.6}$$

Example: Fraction of Executed Instructions is $1024/2432 \approx 0.42$ for the running example.

Number of Divergent Branches ③ Divergent branches (d) are essential to determine *Branch Efficiency*. This measure counts the distinct execution paths of threads during runtime.

Calculation: To calculate this *Number of Divergent Branches*, HyPA stores all traces during the simulation and determines a distinct set of these traces. *Example:* The *Number of Divergent Branches* is 2 in the case of the running example.

Branch Efficiency (η) ③ is an indicator for the utilisation of GPU. A GPU is utilized efficiently if all branches are executing the same instructions. *Calculation:* The *Branch Efficiency* is defined as the ratio between the *Number of Branches* (b) and the *Number of Divergent Branches* (d) and can be calculated as follows [116]:

$$\eta = \frac{b - (d - 1)}{b} \tag{6.7}$$

If there are no divergent branches, the efficiency is 1; if every branch is a divergent branch, the efficiency is 0. *Example:* $Branch\ Efficiency \approx 99.22$ in the running example.

6.2.4 Profile Generation

After the analysis, a PTX profile in the form of a CSV file is generated. This file can be used for power and performance prediction and other optimization tools. All analyzed PTX files are combined into a single output file, which includes details such as filename, PTX version, PTX target, PTX address size, file instructions, CTAs, thread count, static instructions count, dynamic instruction count, FP instructions count, executed instructions count, divergent branches, divergent branches for each CTA, branch efficiency, and duration (in milliseconds).

6.3 Experimental Results

To evaluate the hypothesis that the hybrid approach combines the advantages of static analysis and simulation approaches while avoiding their disadvantages, the following three research questions aim to answer our evaluation.

Research Question 6.1. *How does the run-time of the profiling approach compare to the hybrid approach?*

Research Question 6.2. *How much do the acquired metrics of the hybrid approach differ from the profiling approach?*

Research Question 6.3. *Can the hybrid approach measure metrics that the static analysis can not, and how do they compare to the profiling approach?*

The research questions are based on the profiling approach instead of the simulation approach due to three reasons: (1) The profiling approach is the most accurate, i.e., it surpasses the simulation approach, (2) it is faster than the simulation approach and thus more suited for the evaluation, and (3) it does not affect statements about the simulation approach if evaluated in favor of the hybrid approach. The third reason holds because the profiling is faster than the simulation, i.e., if the hybrid approach is faster than the profiling approach, it is faster and more accurate.

To evaluate these questions, the hybrid dynamic code analysis is performed on several different CNNs ($n = 32$), and the results are compared against classic profilers. Following the technical setup is presented. It uses benchmarks and computed metrics; the evaluation results will be presented afterward.

Technical Setup

The technical setup is identical to those in chapter 5: A SLURM-based HPC cluster, and it is ensured that the same cluster machine is used in all experiments. The used machine is equipped with three NVIDIA V100S 32GB, 256GB memory, and 2 AMD EPYC ROME 7272. The home directory is a network-attached storage connected by a 10GBit/s ethernet connection. Besides this, a second system executes HyPA; it runs on a Lenovo ThinkPad T490s with Intel i7-8565U, 16GB memory, and Ubuntu 22.04.

Benchmark CNNs

The following overviews the CNNs used for all experiments. They differ in various aspects, like the number of layers, neurons, or input layer size. In Table 2.1, the

CNNs and their attributes are listed.

Moreover, different CNN designed for different use cases are considered for the benchmarks. While some CNNs are designed to reach the best prediction accuracy, like Resnet [13], Alexnet [63], or Densenet [117], some are designed to perform well on mobile devices like MobileNet [118]. In contrast, the NASnet-mobile and NASnetlarge are designed by NAS [66] techniques. All CNNs are pre-trained and downloaded from Tensorflow Hub as the analysis focuses on the inferencing aspect of these networks.

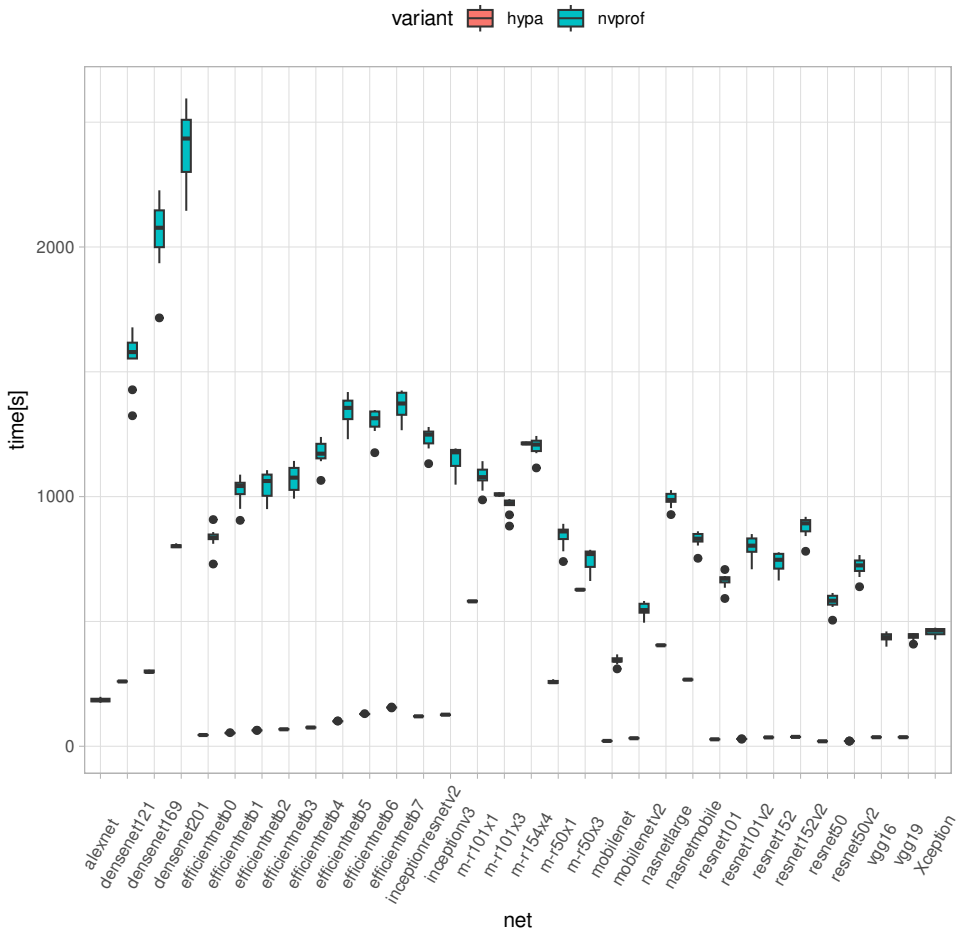


Figure 6.3: Run-time comparison for HyPA and nvprof

Data Generation

For each CNN, two profiles are computed: One, as a result of applying HyPA, and one, applying nvprof as the reference value. To determine the average execution time of both HyPA and nvprof for the profile generation, each profile generation is repeated 10 times. The resulting metrics (besides run-time) are identical for all runs, as both approaches are deterministic.

Experimental Results

Figure 6.3 illustrates the runtime results for both HyPA and nvprof as boxplot comparisons. For every net, the right boxplot shows the nvprof results, and the left boxplot shows the HyPA results. The results of HyPA are smaller and show less variance than the results of nvprof. This decreases the visibility of the boxplots' colors. Therefore, Figure 6.4 visualizes this relation again, but with focus on the pairwise comparison of results. The different colors indicate different CNNs, while the line depicts the bisector. Values above the bisector indicate greater runtimes for HyPA, values below the bisector show greater runtimes for nvprof. Most nets have shorter runtimes for HyPA than for nvprof. The results for the execution time yield mean values between $185.3s$ and $2410.1s$ for nvprof, mean values between $19.63s$ and $1220.1s$ for HyPA. For 27 out of 32 CNNs, the reduction in runtime is statistically significant ($\alpha < 0.05$, Welsh's t-test for comparing means).

Regarding the acquired metrics, our experiments are illustrated in Table 6.1. It shows values for the instruction count results, counts of floating-point operations, and results for branch efficiency. Additionally, the percentage of instructions that need execution (simulation) for HyPA are reported. Results for both types of instructions show varying results. In most cases, HyPA reports a smaller instruction count than the static analysis, possibly due to the conservative overestimation of static analyses. For seven CNNs, the HyPA reports a higher instruction count than static analysis. The instruction count results for nvprof are significantly higher than both HyPA and static analysis, which will be discussed in the next section.

As HyPA considers runtime information, it can report metrics such as branch efficiency, which is impossible for static analysis by design. The reported values of HyPA for branch efficiency deviate by 0.5 - 7 percentage points from the ones

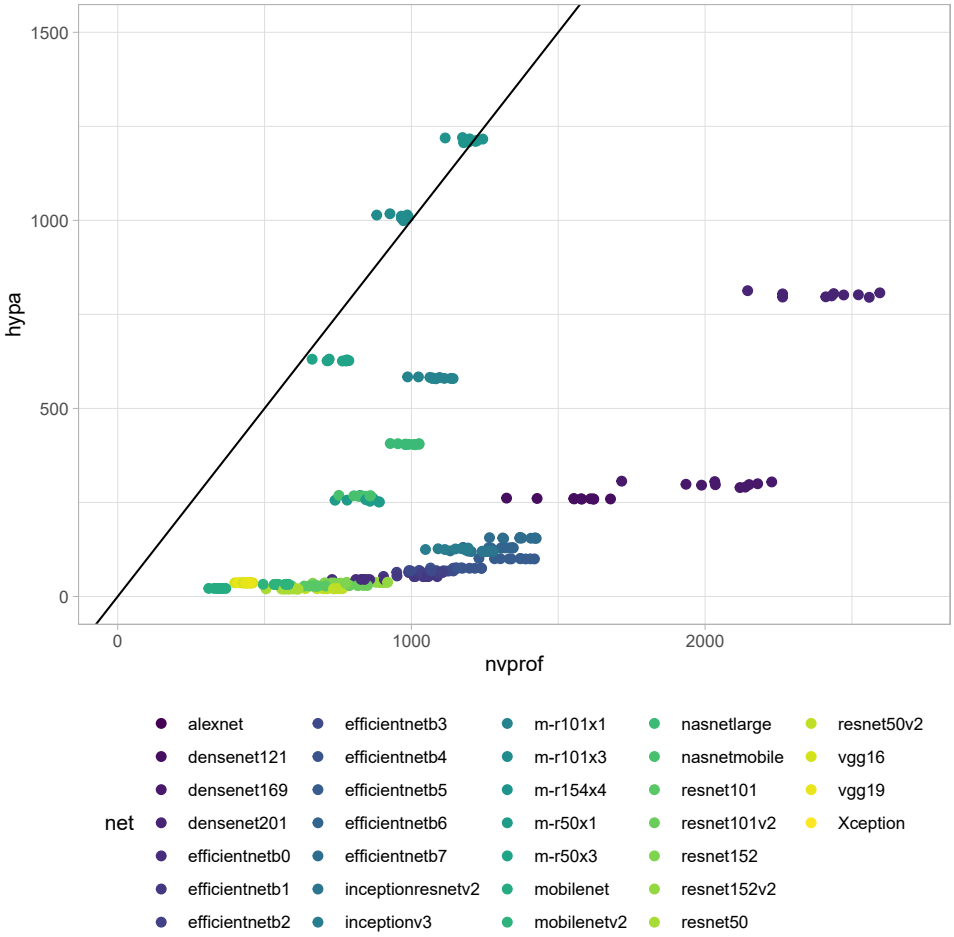


Figure 6.4: Run-time comparison for HyPa and nvprof in correlation

reported by nvprof, with one outlier that deviates by 12 percentage points.

Overall, HyPA had to perform at most 10 % of the instructions to obtain the necessary information to compute its required metrics. This is in line with the runtime results described above.

6.4 Discussion

Following, the interpretations of the evaluation and the current limitations of HyPA are presented and discussed.

6.4.1 Evaluation interpretation

Research Question 6.1 run-time comparison: it is been found that HyPA is faster than nvprof in 27 out of 32 experiments. Therefore, it can be assumed that HyPA is also faster than classical simulation, such as GPGPU-Sim, since a simulation is slower than profilers like nvprof.
Research Question 6.2 how does the metrics differ: it is been found a significant difference between the calculated instructions by HyPA and the actual values measured by nvprof. Three reasons can explain this difference:

1. PTX instructions are not directly translated one-to-one into SASS instructions, and since NVIDIA does not provide documentation on SASS, it is difficult to understand the translation process [78].

2. During the translation process, further optimization steps can be performed, resulting in one PTX instruction being translated into multiple SASS instructions [78].

3. CUDA kernels can be called and executed multiple times, which is not evident in the PTX code but can be observed in the nvprof profiles.

Comparing nvprof metrics like *instruction_executed* and *flops_count_sp* can be complicated because some work on wrap-level, while others work on thread-level [78].

Verifying points 1 and 2 through reverse engineering is complicated. However, verifying point 3 is possible by simulating parts of the CUDA code to obtain the exact number of kernel calls. Therefore, integrating HyPA into a compiler such as Low Level Virtual Machine (LLVM) or Clang can help with this step. It is essential to apply the concept of HyPA to both CUDA and PTX, not just PTX.
Research Question 6.3 measure more metrics: can be answered and demonstrate that HyPA can gather metrics beyond what static code analysis can provide, such as branch efficiency. Additionally, HyPA can accurately predict branch efficiency with a high degree of accuracy, ranging from 0.5 to 7 percentage points off from what is reported by nvprof. When combined with the speedup, HyPA represents a critical improvement for power and performance estimation, as well as the extraction of necessary metrics.

6.4.2 Limitations

Besides all its merits, HyPA still has some drawbacks and limitations. Currently, two aspects limit our approach 1) not all PTX instructions are implemented, and 2) data dependencies in conditional jumps might not resolve correctly. In the following, each limitation is described in-depth and a possible solution to overcome these.

Since not all PTX instructions are implemented (e.g., indirect addressing), our HyPA implementation has to fall back on state-of-the-art approaches. Thus, the results might still be approximated. In future work, it is been planned to implement an equivalent for all PTX instruction and, thus, overcome this issue.

Regarding the second limitation: In some cases, a register is written in two parallel conditional jumps. Currently, HyPA adds an edge to the first node in the dependency graph where a source register is written. This might lead to a wrong control flow when the register is written in both branches. This issue can be solved by implementing an Static Single Assignment (SSA) Form [119, 120] or by earlier dynamic analysis and determining which branch is to be considered. Thus, the edge for the dependency can be added between the correct nodes instead of the earliest occurring node that writes to a register and thus generates a data dependency. We plan to modify the dependency graph generation and extend it by control flow generation with SSA Form to overcome this issue and improve HyPA to get closer to the most accurate calculation of the total number of instructions.

6.5 Conclusion

A novel tool is presented that can analyze PTX code without running it on a real GPGPU. To account for divergent branches, HyPA uses partial execution and code emulation. By reimplementing the PTX ISA in C++, HyPA can emulate the execution on a CPU system. Unlike traditional GPGPU simulators that run the entire application, HyPA only executes the necessary parts to identify divergent branches. This speeds up the analysis, and the execution time is comparable to or faster than profiling with nvprof on the NVIDIA V100S.

The chapter shows that HyPA can provide metrics beyond what static code analysis can offer, such as branch efficiency. One significant advantage of HyPA is its ability to speed up the design space exploration process by providing early metrics for different approaches to power and performance prediction of CNNs on GPGPUs. HyPA's quick execution time makes it an excellent tool for this type of work.

Although HyPA is capable of parsing the entire PTX ISA, it currently has a limitation in its ability to emulate indirect addressing. As part of future work, it is aimed to implement indirect addressing in order to fully cover the ISA emulation.

Table 6.1: Experimental results for static and dynamic analysis with HyPA for 32 different CNNs.

	NVIDIA V100s									
		Instruction Count						Branch Efficiency		
CNN	**% to execute**	**HyPA**	**Static**	**ratio**	**nvprof Instruction**	**nvprof FLOPs**	**HyPA FLOPs**	$\jmath_{nvprof}$	$\jmath_{HyPA}$	ratio
densenet121	9.46%	3,347,534	10,722,398	0.3122	73,918,277,390	1,239,313,864,022	340,755	92.1568%	98.9158%	0.9316
densenet169	9.76%	4,618,062	19,331,166	0.2388	132,677,813,819	2,323,797,491,957	392,979	92.1568%	99.0541%	0.9303
densenet201	9.86%	6,975,950	26,674,398	0.2615	268,813,644,433	4,853,604,008,393	478,995	92.1568%	99.1078%	0.9298
efficientnetb0	7.21%	1,605,049	3,563,417	0.4504	28,170,395,486	341,116,792,220	356,583	92.1535%	96.0827%	0.9591
efficientnetb1	6.24%	2,284,762	4,870,436	0.4691	44,518,197,354	617,624,476,202	568,808	92.1535%	96.2471%	0.9574
efficientnetb2	5.76%	3,234,232	6,097,016	0.5304	74,094,157,251	824,407,223,138	776,827	92.1535%	96.3291%	0.9566
efficientnetb3	6.40%	3,785,200	6,140,976	0.6163	110,536,344,626	1,103,881,214,923	894,237	92.1535%	96.8977%	0.9510
efficientnetb4	6.04%	4,167,634	6,454,880	0.6456	128,613,616,058	1,502,321,376,491	1,014,889	92.1535%	97.2130%	0.9479
efficientnetb5	5.57%	6,988,730	10,670,869	0.6549	154,166,225,231	1,999,289,837,215	1,691,145	92.1535%	97.5638%	0.9445
efficientnetb6	6.13%	8,311,211	12,591,814	0.6600	133,278,295,936	2,212,260,957,075	2,037,615	92.1535%	97.9676%	0.9406
efficientnetb7	6.28%	9,979,998	13,901,223	0.7179	98,031,779,822	2,251,834,391,082	2,362,090	92.1535%	97.6439%	0.9437
InceptionResNetV2	7.89%	4,986,406	9,935,678	0.5018	128,538,060,350	2,591,083,345,663	666,852	92.1568%	99.1184%	0.9297
inceptionv3	7.06%	3,591,286	6,674,334	0.5380	99,139,487,866	1,709,094,946,347	429,036	92.1568%	98.6580%	0.9341
mobilenet	10.77%	792,566	1,077,652	0.7354	15,516,090,366	287,752,628,102	169,807	92.1086%	97.3356%	0.9462
MobileNetV2	10.22%	995,962	2,032,606	0.4899	15,757,543,754	210,300,655,245	177,403	92.1538%	98.2702%	0.9377
m-r101x1	7.11%	96,990,375	28,257,288	3.4324	45,603,141,589	870,125,486,565	10,790,977	92.1568%	98.6123%	0.9345
m-r101x3	8.43%	187,338,305	42,787,720	4.3783	101,530,724,333	2,734,139,342,013	21,533,361	99.6015%	99.1282%	1.0047
m-r154x4	9.53%	267,066,941	62,232,168	4.2914	156,582,406,489	4,138,138,061,309	34,877,039	99.6015%	99.2344%	1.0036
m-r50x1	8.28%	46,505,966	14,963,624	3.1079	44,031,032,572	853,827,859,109	5,338,521	92.1568%	98.5723%	0.9349
m-r50x3	9.02%	90,945,907	20,466,728	4.4435	91,364,961,407	2,589,336,187,773	10,241,279	99.6010%	99.0358%	1.0057
nasnetlarge	7.55%	5,776,912	28,546,814	0.2023	76,928,601,211	1,702,504,253,055	568,407	92.1566%	99.2786%	0.9282
nasnetmobile	8.31%	3,363,489	19,124,047	0.1758	8,651,631,107	103,056,406,909	192,899	92.1536%	99.1792%	0.9291
resnet101	5.77%	2,728,871	2,922,782	0.9336	33,718,778,616	665,107,865,206	625,983	92.1565%	98.8248%	0.9325
resnet101v2	6.13%	2,804,054	3,035,678	0.9236	53,668,419,497	1,109,393,289,019	570,448	92.1565%	98.8290%	0.9324
resnet152	5.50%	3,923,239	4,117,022	0.9529	34,822,490,986	680,214,892,914	917,567	92.1565%	99.0168%	0.9307
resnet152v2	5.83%	4,012,758	4,248,094	0.9446	54,769,117,374	1,124,161,761,326	864,080	92.1565%	99.0196%	0.9306
resnet50	6.51%	1,532,071	1,725,982	0.8876	32,615,069,876	650,429,295,715	334,399	92.1565%	98.2734%	0.9377
resnet50v2	6.88%	1,605,974	1,832,990	0.9761	52,567,694,245	1,094,624,830,407	280,912	92.1565%	98.2815%	0.9376
vgg16	9.56%	8,738,852	412,254	21.1977	96,577,294,513	1,480,562,789,621	1,292,884	92.0827%	86.9564%	1.0589
vgg19	9.56%	8,738,852	412,254	21.1977	94,645,096,385	1,488,245,837,048	1,292,884	92.0827%	86.9564%	1.0589
Xception	6.37%	2,223,811	3,402,190	0.6536	24,125,255,065	735,441,505,983	391,266	76.3605%	98.4047%	0.7759

7 Performance Estimation of CNNs for GPGPUs

The chapter presents a novel automated approach, enabling designers to fast and accurately estimate the performance of CNNs for GPGPUs in the early stage of the design process. The proposed approach uses static analysis for feature extraction and Decision Tree regression analysis for the performance estimation model. Experimental results demonstrate that our approach can predict CNNs performance with an absolute percentage error of 5.73% compared to the actual hardware. The presented research is based on the previously published work [9].

The chapter is structured as follows: A brief introduction is given in Section 7.1. Afterward, the methodology is presented in Section 7.2, followed by the experimental results in Section 7.3. The results are discussed in Section 7.4. Finally, the chapter closes with a conclusion in Section 7.5.

7.1 Introduction

In general, the DL life-cycle has two main phases, which are 1) training, where the DL models are trained based on the training data set, and 2) inferencing, where the finally trained DL models (e.g. a CNN) are executed on real hardware. Since the training phase is offline, high-performance computing devices (e.g., powerful data centers GPGPUs like the NVIDIA V100 and NVIDIA A100) are usually used without restrictions on non-functional design aspects. In contrast, the inferencing phase is an online process where the non-functional design aspects are vital in defining the overall design constraints. For example, in the case of (small) IoT devices, the selected accelerator directly impacts the design constraints such as

low latency and cost of the final product [121].

Hence, selecting the right DL accelerator (e.g., GPGPUs) in this phase is of utmost importance to perform on-time computations (e.g., in the case of autonomous driving) and meet design constraints. Without a fast and automated approach, designers must build several prototypes and test numerous hardware platforms to find the right accelerator for the inferencing phase, which is very time- and cost-intensive.

To overcome this issue, several automated methods have been developed that can be divided into two main categories: GPGPU simulators [112, 113] and ML-based estimation methods [39, 43, 101]. GPGPU simulators such as GPGPU-Sim [112], or GPU-ocelot [113] are usually used to perform DSE and obtain the performance of a given application without the need for actual hardware execution. They use a combination of performance counters and specific hardware details to measure the performance of applications. The obtained results have an accuracy between 10% to 20% compared to the actual hardware execution [39]. However, these simulators require a significant execution time to obtain the results and thus are significantly slower than actual hardware execution. On the other hand, ML-based estimation methods provide designers with a fast solution to obtain the design parameters of a given application, such as performance. However, they either require specific performance counters, kernel settings [39, 43], or detailed platform descriptions and the scheduling of different CNNs operators on different platform processing [101], which may only sometimes be available.

This Chapter focuses on the performance estimation of CNNs for GPGPUs, one of the most popular DL models in various domains. A novel approach is presented, allowing designers to predict a given CNN's performance for GPGPUs quickly. In contrast to the existing methods that rely on specific setups (performance counters or kernel settings), the proposed approach is developed based on a simple ML model and easy-to-extract features, namely CNNs number of trainable parameters, number of PTX instructions, and GPGPUs architectural information.

The experimental results demonstrate the effectiveness of our approach in estimating the performance of CNNs for GPGPUs where a MAPE of 5.73% with an R^2 of 0.45 and an adjusted R^2 of 0.19 in comparison to the actual hardware execution

is obtained. Moreover, our proposed approach is significantly faster.

In summary, the main contributions of this paper are as follows:

- proposing a quick and highly accurate performance prediction model of CNNs for GPGPUs with a minimal dependency on the runtime performance counter compared to the state-of-the-art methods. The proposed approach has no runtime dependency for the final prediction,
- supporting the cross-platform prediction due to the consideration of hardware features,
- comparing different ML algorithms to obtain the best performance predictive model (i.e., Decision Tree regression analysis),
- evaluating the proposed approach on estimating the performance of different standard CNNs for various GPGPUs such as NVIDIA 1080Ti, and V100S.

7.2 Methodology

The proposed methodology comprises two main phases illustrated in Fig. 7.1. Each phase of the proposed approach is explained in more detail.

7.2.1 Training Dataset Creation

The performance (accurate number of Instruction per Cycle (IPC)) of a given CNN running on a GPGPU is significantly related to two main factors: the architectural features of the GPGPU and the complexity of the CNN model. The architectural features (CUDA cores, memory, registers, or L2 cache) of a GPGPU are defined based on the type, size, and number of components the GPGPU consists of. Hence, the performance of a CNN running on different GPGPUs is not identical and varies. These architectural features of GPGPUs that impact the required number of IPC when a CNN runs on them were extracted for the training dataset. When the training dataset is built, this information is used as a predictor

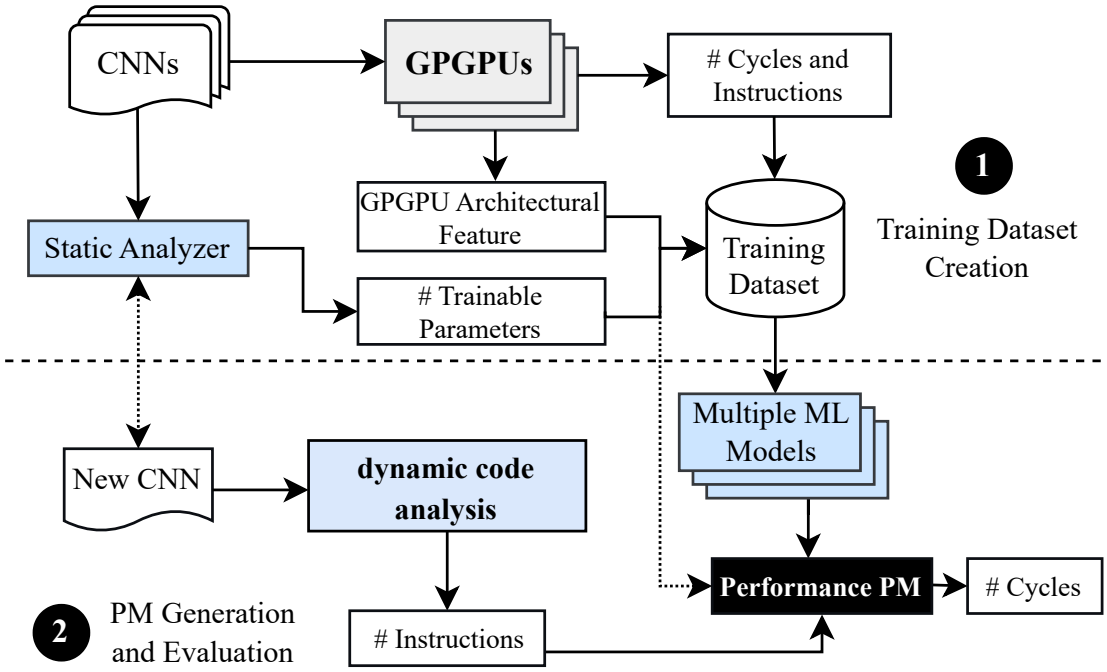


Figure 7.1: Performance estimation methodology.

(inputs). This work considers two different GPGPUs, the NVIDIA V100S and the NVIDIA GTX 1080Ti, for the training phase. They have different specifications and architectures to cover a variety of features.

The number of trainable parameters and the total number of PTX instructions can specify the complexity of NNs. A trainable parameter is a weighted connection between the neurons, meaning more trainable parameters need more calculations to produce the final output. To obtain the trainable parameters for a given CNN, a static analysis is performed using the *Static Analyzer* module (Fig. 7.1), where first, the trainable parameters for each convolutional layer are calculated. Next, based on the number of layers, the total number of trainable parameters for the CNN is achieved.

The *Static Analyzer* module performs all calculations for all 32 CNNs used during the first phase illustrated in Fig. 7.1. The results of the *Static Analyzer* module are unified with all other predictors in the training dataset. Modern frameworks like Pytorch and Tensorflow supply functions that can be used to calculate the

trainable parameter quickly and accurately.

Table 2.1 in chapter 2 provides an overview of the CNNs used for all experiments. They differ in various aspects, like the number of layers, neurons, or input layer size. As the table shows, most CNNs have the same input size. That is because most are trained on the ImageNet data set. However, it is been ensured that also CNNs with different input sizes are considered.

HyPA is used to generate the total number of executed PTX instructions. By this, the approach is overcoming the lack of speed of a traditional simulator since HyPA only executes a small part of the code (see chapter 6). Moreover, this enables our approach to calculate the total number of PTX instructions for any CNN without executing it on an actual GPGPU. The total number of PTX instructions – calculated by the dynamic code analysis – is used as predictors (inputs) for the training data set.

By running CNNs on GPGPUs, the accurate number of IPC can be obtained and is considered as the training dataset's response (output). Moreover, all of the 32 test CNNs are executed on different GPGPUs while measuring the number of IPC with the nvprof profiler provided by NVIDIA. It is been ensured that the 32 CNNs used for the dataset generation have different complexities and sizes.

An item of the final training dataset D is formally denoted by a vector:

$$d = (y, p, c_1, .., c_m, t) \tag{7.1}$$

Where for each observation d, the input parameters p, $c_i, 1 \leq i \leq m$, t identify the total number of instructions, the GPGPU architectural features, and the total number of CNN trainable parameters, respectively. The output parameter y denotes the measured performance (number of IPC) of each CNN running on GPGPUs. D is the set of all measured data points building the training data. The training data set D is split into D_t containing 70% of the data points for training and D_v containing 30% of all data points for evaluation. Consequently, all evaluation data points are completely new to the trained model.

7.2.2 Predictive Model Generation and Evaluation

Five different standard ML algorithms are evaluated, namely *Decision Tree*, K-NN, *Random Forest Trees*, *Linear Regression* and XGBoost. ML aims to locate patterns in the given set that provide the most straightforward explanation possible of the phenomenon. That follows Occam's razor, which states that if several theories explain a given phenomenon, the one making the least assumptions probably is the right one [122].

The Linear Regression is selected to justify if there are linear dependencies between the output (e.g., number of IPC) and the predictors. Moreover, K-NN and Decision Tree regressions are selected to consider algorithms that can show non-linear dependencies. Since the Random Forest Tree is an ensemble of Decision Trees, it is considered as an advanced method of the Decision Tree. Since the execution time of the predictive model is important to speed up the DSE, the XGBoost is taken into account, a frequently used boosting system, to improve execution time and accuracy of tree classifications and regressions [106]. Furthermore, the runtime of K-NN is significant depending on the dataset since the execution time increases linearly proportional to the number of data entries in the training data set, which can cause the necessity of faster techniques. As shown in chapter 4, the predictive model based on NN is only negligibly better performing than the K-NN on power prediction. Since the training dataset is small, NNs are not considered for prediction as larger training datasets are usually required.

As shown in Fig. 7.1, for a given new CNN (in the evaluation step of the second phase), the GPGPU architectural features and the CNN trainable parameters (inputs of the predictive model) are extracted using the *Static Analyzer* module. The total number of PTX instructions is extracted from abstract PTX files by the dynamic code analysis module (as the PTX contains dynamic information such as the length of loops or jump instructions based on the comparison). Consequently, no execution of the CNN on real hardware or cycle-level simulators (which take much longer than real devices) is required.

The experimental results are evaluated using the MAPE and the R^2 coefficient. That also enables us to compare our results to state-of-the-art research. It returns a value between 0 and 1. An R^2 near 1 means a high fit between the model and

underlying data. A negative value indicates a bad fit between the model and data.

7.2.3 Differentiation of Methodologies

The methodology in this chapter is based on the methodologies presented in chapters 4 and 5 but differs from these in two key respects.

1) Instead of power consumption, the number of cycles required for the calculation is measured for this methodology and used as a label for the prediction. The resulting data set has a similar structure to Chapter 5, so the two data sets can, in principle, be combined into one.

2) To overcome the disadvantages of collecting the number of instructions from chapters 4 and 5, namely that static code analysis needs to be more precise and nvprof requires a single execution on a referential GPGPU, the HyPA presented in chapter 6 was used. This allows the number of instructions to be determined based on the PTX code. Compared to static code analysis, HyPA considers the dynamic runtime dependencies and resolves these by partially simulating the code. This means that complete execution on a reference GPGPU is not required, and greater accuracy is achieved than static code analysis.

Table 7.1: Comparison of four different ML-regression algorithms in terms of accuracy and execution time

Regression Model	MAPE	R^2	adj. R^2
Linear Regression	8.07%	-0.0034	-0.4439
K-NN	5.94%	0.34	0.08
Random Forest Tree	7.12%	0.22	-0.12
Decision Tree	5.73%	0.45	0.19
XGBoost	7.59%	0.14	-0.24

7.3 Experimental Results

Following the experimental results are illustrated. The experiments aim to answer the following research questions:

Research Question 7.1. *Which ML approach has the best prediction results?*

Research Question 7.2. *Which are the most important features used by the best predictive model?*

Research Question 7.3. *Compared to a naive approach, which speed-up can be achieved for DSE?*

Our experimental results demonstrate that the performance prediction based on GPGPU architectural features, the number of CNN instructions, and trainable parameters is promising. Five ML-algorithms are evaluated: Linear Regression, K-NN, Random Forest Tress, Decision Tree, and XGBoost. Table 7.1 illustrates an overview of the experimental results. The Linear Regression achieves the worst results with a MAPE of 8.07% followed by XGBoost with a MAPE of 7.59%. The Random Forest Tree, the K-NN, and the Decision Tree are close with a MAPE of 7.12%, 5.94%, and 5.73%, respectively.

Table 7.2: Predictors descriptions used by the decision tree

Features	Brief description	Importance
Executed Instructions	Number of instruction to be executed	0.0141
trainable params	Number connections between neurons	0.2599
Memory Bandwidth	Available memory bandwidth	0.72583

The $R2$ and adjusted $R2$ of the linear regression indicate no linear dependencies between output and predictors. Another interesting point in this experiment is that the results of the Decision Tree are better than those of the Random Forest Trees. The main reason could be that the decision is based on the average value of all included decision trees for random forest trees. Therefore, the results could be distorted if decision trees exist with poor accuracy.

All regression models considering nonlinear dependencies show promising results based on the experimental results. Due to the results obtained, the best option for building the predictive model is based on the decision tree algorithm. However, these results can be improved by considering a more extensive range of GPGPUs for generating training data sets.

The Figures 7.2, 7.3, 7.4 and 7.5 show the predicted and original performance of six randomly selected standard CNNs [63, 67, 123] (which are entirely independent of the training phase) on our final Decision Tree (predictive model). The results for the Decision Tree are illustrated in 7.2, for the K-NN in 7.3, for the XGBoost in 7.4 and the Random Forest Tree in 7.5. As the figures illustrate, all predictive models' predictions are close to each other and do not differ significantly. Compared to the real hardware, namely NVIDIA GTX 1080Ti, the proposed approach achieves a MAPE of 5.73%, and in the best case, the exact IPC value is predicted by the Decision Tree for the EfficientNetB7. Table 7.2 reports the three predictors with the most impact on the model. The decision tree predictors are chosen based on performing the Gini coefficient during the predictive model training phase. As shown in this table 7.2, only one GPGPU architectural predictor is used, i.e. Memory Bandwidth, and two CNN-related predictors, i.e. number of executed instructions and trainable parameters. Based on our analysis, the Memory Bandwidth has the highest impact on estimating the number of cycles.

Compared to the recent method presented in [124] with a MAPE of 14.73%, our proposed approach provides designers with 2.5 times better accuracy. This comparison also shows that performance prediction based on CNNs topology, the number of instructions, and GPGPU architectural information with the Decision Tree algorithm achieves better results than [124]. Moreover, as [124, 125] do not consider hardware details as features for prediction, they cannot perform cross-platform estimation. Therefore, their models are limited to a single GPGPU.

Assume a DSE scenario (as an application of the proposed approach) where the goal is to obtain the performance of a given CNN for n GPGPUs, the execution time of the proposed approach is defined as $T_{est} = t_{dca} + (n \cdot t_{pm})$ where t_{dca} and t_{pm} denote the time for our dynamic code analysis and execution time of predictive model, respectively. In contrast, the total time with real GPGPUs to obtain similar results (naive approach) is defined as $T_{measur} = t_p \cdot n$ where t_p denote the profiling time (e.g., with nvprof). Since both t_{pm} and t_{dca} are smaller than t_p (seconds vs minutes), T_{est} is in most cases almost equal to t_p or smaller. In this case, compared to the execution time of the naive approach T_{measur}, the proposed approach is significantly faster. That enables designers to estimate the

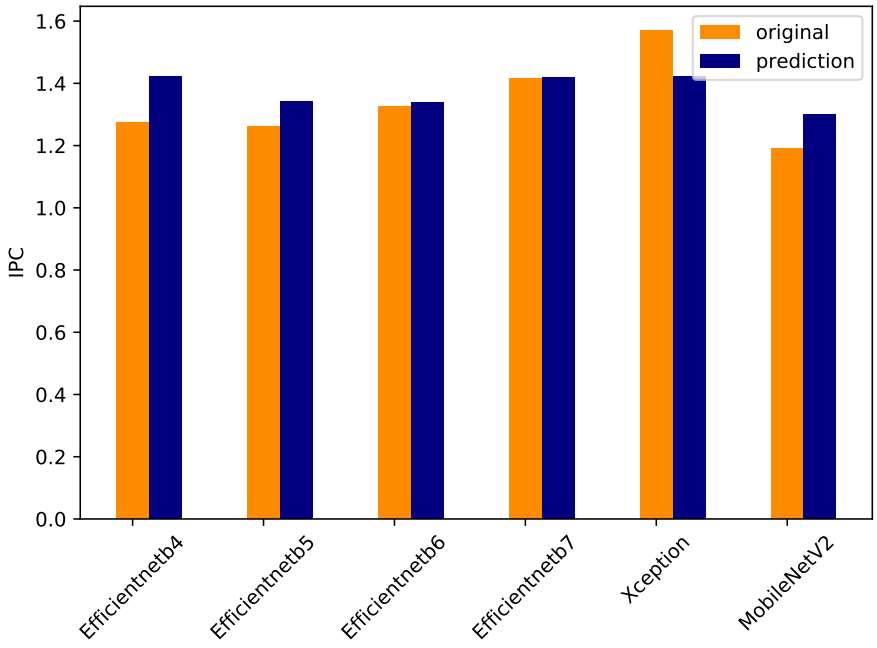


Figure 7.2: Decision Tree predicted

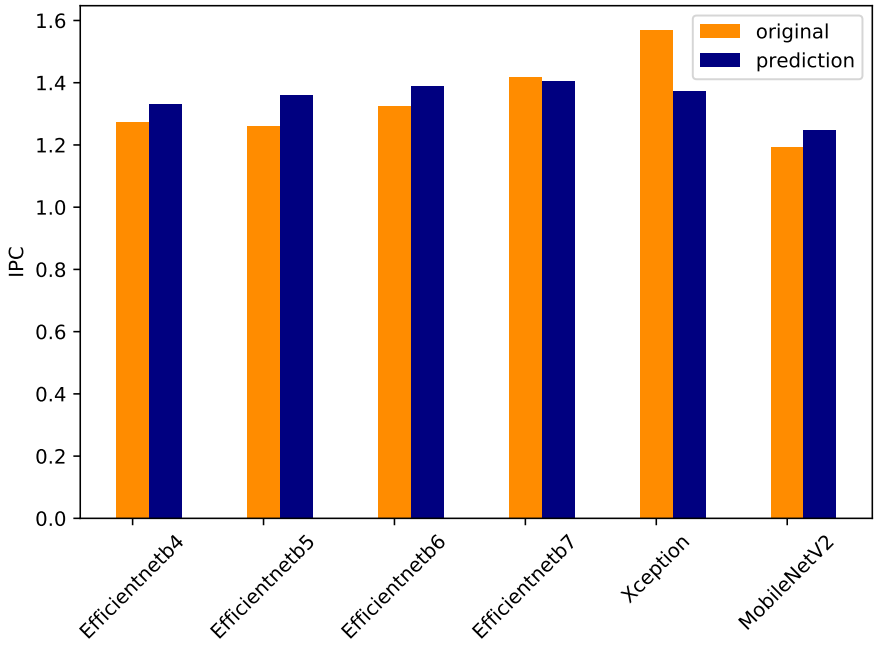


Figure 7.3: KNN predicted

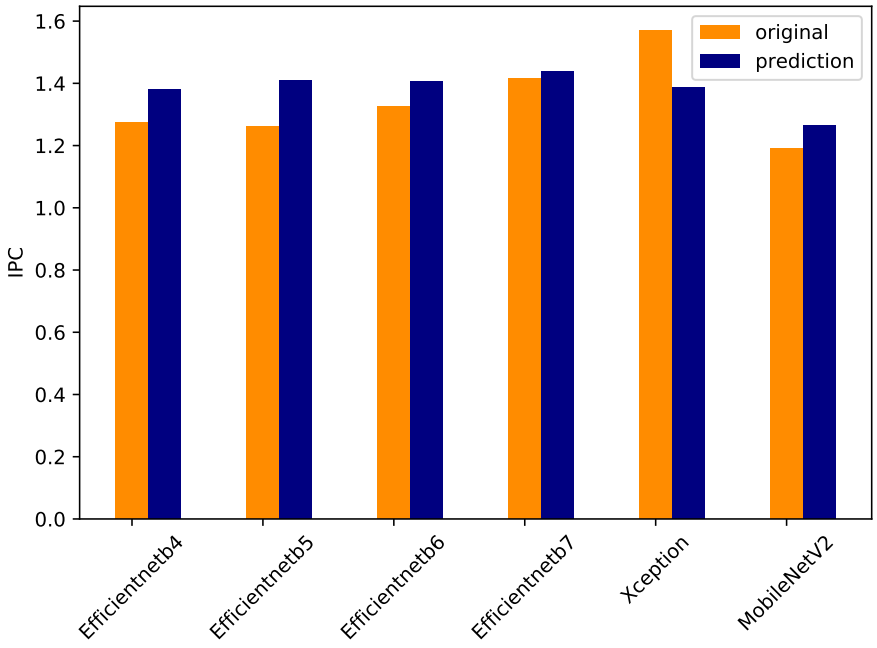


Figure 7.4: Gradient Boosting predicted

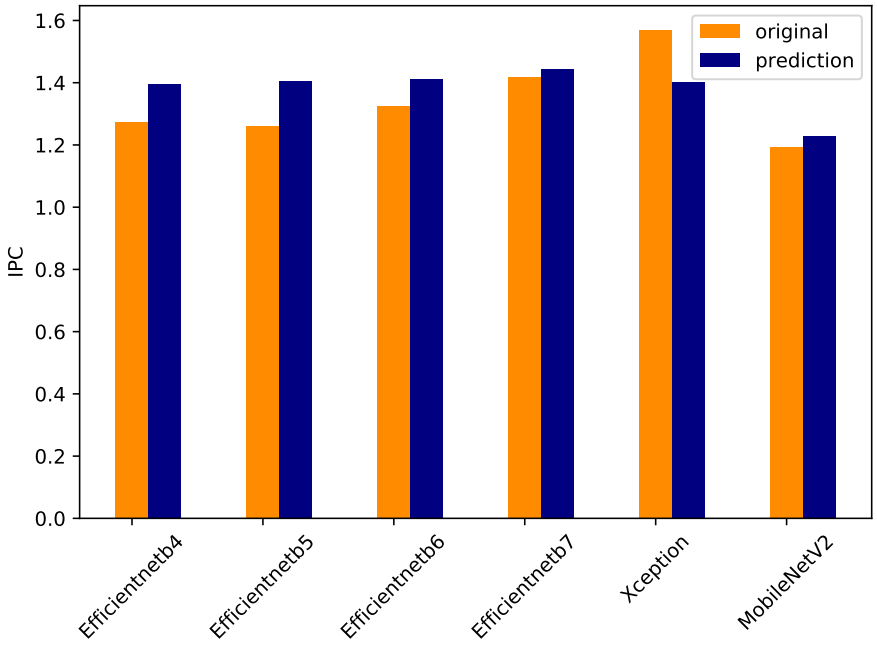


Figure 7.5: Random Forest Tree predicted

performance of a given CNN at the early stages and perform a fast DSE. To prove the concept, Table 7.3 illustrates the measured execution time of seven standard CNNs and their profiling with nvprof on seven different GPGPUs (e.g., NVIDIA GTX 1080Ti, NVIDIA V100S, and NVIDIA Quadro P1000) for the naive approach and our proposed approach. Our approach achieves an average speedup of 33 times for one single GPGPU. The speedup is even higher for larger numbers n of GPGPUs.

Table 7.3: Execution time comparison of the proposed approach versus the naive approach for eight different CNNs in the case of various GPGPUs

CNN	Naive Approach (s)								Novel Approach (s)							
	t_p	n=1	n=2	n=3	n=4	n=5	n=6	n=7	t_{pm}	t_{dca}	n=1	n=2	n=3	n=4	n=5	n=6	n=7
efficientnet b3	663	663	1,326	1,989	2,652	3,315	3,978	4,641	11	24.8	35.8	46.8	57.0	68.8	79.8	90.8	101.8
efficientnet b4	778	778	1,556	2,334	3,112	3,890	4,668	5,446	9	24.0	33.0	42.0	51.0	60.0	69.0	78.0	87.0
efficientnet b5	950	950	1,900	2,850	3,800	4,750	5,700	6,610	8	40.3	48.3	56.3	64.3	72.3	80.3	88.3	96.3
efficientnet b6	936	936	1,872	2,808	3,768	4,680	5,616	6,552	8	60.2	68.2	76.2	84.2	92.2	100.2	108.2	116.2
efficientnet b7	1,037	1,037	2,074	3,111	4,148	5,185	6,222	7,259	1	6.8	7.8	8.8	9.8	10.8	11.8	12.8	13.8
Xception	314	314	628	942	1,256	1,570	1,884	2,198	8	23.6	31.6	39.6	47.6	55.6	63.6	71.6	79.6
MobileNet V2	343	343	686	1,029	1,372	1,715	2,058	2,401	8	12.2	20.2	28.2	36.2	44.2	52.2	60.2	68.2

As the experimental results show, the execution time of the proposed approach T_{est} demonstrates the correctness of the definition above, where the total execution time stays almost the same when the number of GPGPUs increases. This means that, in general, our proposed approach can archive a speed up of n times for n GPGPUs. As a result, the higher the number of GPGPUs for the DSE, the higher the speedup of the proposed approach.

From the Decision Tree, 60 different rules have been derived. Based on the path of the Decision Tree, the rules can be displayed in a disjunctive form. Fig. 7.6 illustrates a small part of the final Decision Tree. Each node represents a test on a feature, while each leaf indicates a class label (decision taken after computing all features) that is the number of IPC. Moreover, branches specify conjunctions of elements that lead to the class label. Hence, the paths from the root to the leaf show classification rules. As an example, the gray nodes can be described using the following rule:

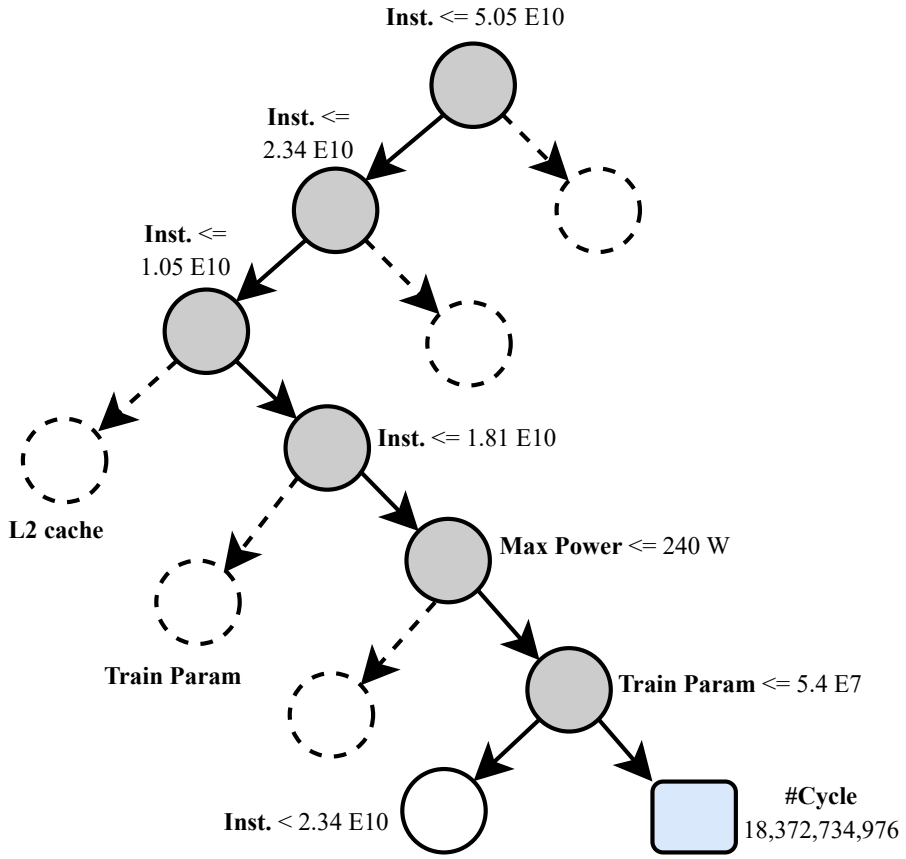


Figure 7.6: Short snippet from the generated decision tree

$$(Inst \geq 5.05E10) \ \wedge \ (Inst > 2.34E10) \ \wedge \ (Inst \leq 1.05E10) \wedge \\ (Inst \leq 1.81E10) \ \wedge \ (MaxPower \leq 240 \ W) \ \wedge \ (Train \ Param \leq 5.4E7) \\ \approx 1.83E10 \ \#Cycles \tag{7.2}$$

Where $Inst, MaxPower, Train\ Param$ stands for the total number of executed instructions, maximum power consumption of the GPGPU, and the total number of trainable parameters of the CNN. This rule (7.2) gives the number of required IPC for a CNN on a GPGPU that fulfills the constraints.

7.4 Discussion

Research Question 7.1 best ML Model:, the experimental results clearly show that the Decision Tree outperforms all other predictive model. However, the difference to the K-NN is negligible. Comparing these results to the aforementioned in chapter 4, it is illustrated that the K-NN performed well for both power and performance estimation of CNNs on GPGPUs.
Research Question 7.2 most important features:, as tab 7.2 illustrates, the most important feature to predict the performance (e.i., number of IPC) is the GPGPU's memory bandwidth, followed by the CNN's number of the trainable parameter and the number of executed instruction counted by HyPA, respectively.
Research Question 7.3 speed-up:, as illustrated in tab. 7.3 significant speedups can be achieved by the presented approach compared to naive DSE. The speedup is increasing with the number of GPGPUs considered for the DSE. Hence, DSE performed with the predictive model for performance estimation is scaling with the number of GPGPUs, making it perfect for large-scale productions.

7.5 Conclusion

This chapter proposes a novel ML-based approach to estimate the performance of CNNs for GPGPUs. It illustrates how the version of a given CNN for GPGPUs can be estimated by analyzing the CNNs topology, instructions, and GPGPUs' architectural information. Experimental results sound promising. Our predictive model achieves a MAPE of 5.73% in performance prediction (accurate number of IPC) compared to actual GPGPUs. Compared to the state-of-the-art methods, the accuracy of our predictive model is up to 2.5 times better. Moreover, it empowers designers to predict the performance of a CNN on different GPGPU architectures without retraining. The proposed approach can also help designers perform NAS with hardware/software co-design to predict the performance of different generated CNN architectures for a wide range of GPGPUs' architectures without the need to execute the CNN on all of them. Hence, using our proposed approach, the DSE process can be sped up significantly.

8 Discussion

The following chapter will discuss the results of Chapters 4, 5, 6, and 7 and bring them into the overall challenges introduced in Chapter 1. Moreover, the initial research questions from Chapter 1 will be answered, these are as follows:

Research Question 1.1 *How is it possible to create rules to support decision-making on whether to execute AI applications on IoT and Edge systems or offload them?*

Research Question 1.2 *How can the power and performance for AI-based applications planned to be executed on GPGPUs be predicted with off-the-shelf techniques without the need for runtime-dependent features?*

Research Question 1.3 *What are the most significant impact factors on the power and performance of AI-based applications executed on GPGPU?*

Research Question 1.4 *Can machine learning-based power and performance estimation speed up the DSE for systems designed for AI-based applications on GPGPUs?*

Research Question 1.1 This thesis presents a statistical analysis of the trade-off between power and performance for AI offloading. Based on the analysis, a decision-making system can be introduced for the experimental setup. By increasing the variation in AI applications and platforms, the decision-making system can be generalized and used for several AI applications and devices. This generalized system will help system designers choose the optimal location for running AI applications. Additionally, this set of rules can be incorporated into AI applications and IoT devices, enabling them to make autonomous decisions about when to offload and when not to.

Research Question 1.2, the power and performance prediction of CNN's inference on GPGPUs is possible. As illustrated in Chapters 4, 5, and 7, the power consumption and performance can be estimated with light way shallow machine learning approaches. As presented, the prediction reached an error of under 6%, and the K-NN algorithm performed well overall. As the prediction relies on only static none-runtime features in Chapters 4 and 7, the presented approaches can be used without actual devices and specific profiler settings that are hard to recreate. As the results of HyPA illustrate, the application does not necessarily have to be executed on an actual device. With the hybrid analysis presented in Chapter 6, it is also possible to consider the runtime dependencies of code execution and thus measure more accurately the number of instructions executed for the CNN. By this, good results for performance predictions are possible. Moreover, the methodology of Chapter 5 – which relies on runtime-dependent features – can simply modified as shown for the performance prediction in Chapter 7 and, thus, also be used without runtime features. Consequently, system designers are enabled to plan prototypes and pre-select AI accelerator (e.i., GPGPUs) during the software development towards a hardware-/software co-design process for AI applications.

Research Question 1.3, certain key features are required for simple power estimation. These include CUDA cores, GPGPU memory, base core frequency, storage speed, GFLOPs, memory clock, L2 Cache, number of return instructions in PTX code, and number of trainable parameters of the CNN. Moreover, the essential features for performance estimation are the number of executed instructions (which can be calculated by HyPA), the number of trainable parameters, and memory bandwidth.

The most critical features illustrate that the number of trainable parameters is crucial in both cases – power consumption and performance – as essential as the executed instruction. As for power consumption, only single classes of instructions are crucial; the total number of executed instructions is vital for performance estimation. Moreover, as the analysis of DFS on GPGPUs illustrates, the core frequency does not significantly impact the performance and execution time. However, the power consumption is increasing with higher core frequencies. Remarkably, when the core frequency reaches higher frequencies, such

as 1200MHz, the power consumption on the NVIDIA V100S increases. Consequently, to run CNNs sustainably on GPGPUs, lower frequencies should be preferred to lower the power consumption as it does not affect the runtime.

Research Question 1.4, it has been shown that the predictive model execution can be significantly faster than the application's profiling on actual devices. Moreover, with HyPA, significant drawbacks of simulations (e.i., speed) can be overcome, and the design of space exploration can speed up without the need for actual devices. Consequently, system designers need to build fewer prototypes, reducing resource waste. That speeds up the prototyping phase and makes it more sustainable.

The thesis's findings indicate that using ML-based predictive models can assist with sustainable system design. This is especially beneficial for designing IoT and Edge devices, which often face limited resources such as battery lifetime, requiring low-power applications and systems. By reducing power consumption while maintaining the performance of AI inference systems, the presented challenges, such as high energy costs in Europe, can be addressed. Low power implementation can help keep European systems competitive with the rest of the world and assist data center providers in complying with Germany's newest energy efficiency laws (e.i., EnEfG). Although the proposed methods are demonstrated on CNNs, they apply to almost all kinds of NN. These methods provide system designers with tools to enhance DSE with ML-based prediction models, allowing them to estimate power consumption or performance at early design stages without running the application on a prototype.

Throughout this thesis, novel approaches for important challenges in sustainable AI computing are developed and presented. Moreover, related work showed that similar approaches also work for general applications and other accelerators like RISC-V [126, 127, 128, 129, 130]. Consequently, a more general model that also could perform cross-platform predictions is an interesting research area for future work. Thus, combining the proposed approaches with those of [130] can lead to cross-platform predictions.

Limitations

As every study faces its limitations, this thesis also has two limitations: 1) inaccuracy of *nvidia-smi* and 2) lack of variation in the applications.

Inaccuracy of *nvidia-smi*: A recent study [131] stated that the power measurement with *nvidia-smi* and the built-in power meter lead to larger errors than documented by NVIDIA. NVIDIA claims an error of $\pm 5W$ while [131] could detect an error of $\pm 5\%$. As the latest GPGPU models draw up to 700W; a 5% Error would lead to $\pm 30W$ instead of $\pm 5W$. Although the error of $\pm 30W$ might appear negligible, it can be accumulated on HPC Data Center with thousands of GPGPUs. Consequently, as the predictive models in this work rely on the power meter and measurement using *nvidia-smi*, the prediction error could be more extensive when compared to measurements with external power meters. The techniques proposed in this thesis can be used independently of the type of power meter used. To obtain more accurate results, system designers can create their own training data sets using external power meters that provide more precise measurements. This requires updating the process of generating training data sets and using different power meter options instead of relying on the built-in power meter and *nvidia-smi* tool. Additionally, not all NVIDIA GPGPUs support the power meter functionality of *nvidia-smi*, so using external power meters can also enable the inclusion of more GPGPU models in the training data set generation process. Therefore, selecting external power meters is a valuable improvement to the methodologies presented in this thesis, and it can further enhance the benefits of predictive models for power consumption prediction. However, for the power measurements of the NVIDIA Jetson Nano in Chapter 3, external power meters were used to verify the internal power meter of the Jetson Nano, and no difference between the external and internal measurements could be detected. Thus, this limitation only affects the larger Data Center GPGPUs.

Lack of variation on application: Only CNNs are used as examples to prove the proposed approaches. Although all approaches are designed to use high-level NN attributes that exist in all kinds of NNs, it can be possible that small modifications are required to include other NN types. This limitation can be overcome by increasing the training data and adding other types of NNs.

9 Conclusion

This thesis presents a statistical model that can be used to derive a decision-supporting system to choose if offloading AI applications to the cloud or edge is a valid option. As the statistical analysis shows, slow networks with low bandwidth and high latency, such as 2G, are a good option in some instances. Power savings were found across the board for all mobile network types. However, the influence CNNs' size, bandwidth, and latency must be considered for performance purposes. Consequently, performance improvement needs careful selection of the mobile network type; otherwise, performance losses must be expected. Implementing the statistical decision-supporting system into the system design workflow will help the system designer to solve the placement problem of AI applications (see challenge 1 in chapter 1). Afterward, the system designer can use predictive models to customize the systems. Therefore, this thesis presents innovative techniques for analyzing and estimating the power consumption and performance of CNN-based applications on GPGPU. The methods are demonstrated on CNNs, but they can be applied to almost all types of DNNs. This is because the approaches consider general architecture features of all kinds of DNNs. The excellent experimental results illustrate the robustness of the formerly presented approaches. In best cases, it is possible to estimate power consumption and performance with a MAPE of 0.088% and 5.73%, respectively. Furthermore, considering the power consumption estimation for DFS improves the DSE time significantly. Hence, the investigation of the design space for the NVIDIA V100S with 196 available frequencies requires only about 64s for 30 different CNNs with the proposed approach from chapter 5. Conversely, when profiling the CNNs and all frequencies with nvprof, it takes about 195 hours. Hence, using the predictive model leads to a rigorous speedup in DSE. The speedup improvement and opportunities for DSE for AI system design are tremendous. Considering predic-

tive models for assembling AI systems will lead to fewer prototypes and a faster design process. Moreover, the presented approaches can already be applied at early software development stages, moving toward a standardized hardware/-software codesign process for AI applications. Additionally, the thesis proposes a new hybrid profiling approach for PTX code called HyPA. This approach extends the static analysis metrics and improves the execution time compared to simulation or profiling on real devices. As illustrated in chapter 7, using HyPA for data gathering for predictive models can further improve the DSE time. System designers can use the combination of HyPA and predictive models to improve their workflow, reduce the number of prototypes they need to build, and decrease time-to-market. Moreover, the profiler can generate the results for both static codes analyzing metrics and simulation-based metrics, enhancing the profiling process even more and giving software and AI developers crucial insights into the application behavior.

Overall, the formerly presented approaches improve and speed up the computer architecture and hardware selection process for ML inference systems and can enhance the DSE process. With the proposed decision-making support system for offloading or local execution, designers can easily decide on functionality placement. Afterward, the system designer can use the different predictive model approaches and estimate the power and performance of different GPGPUs for the system. Consequently, the selection of AI accelerator (i.e., GPGPU) can be reduced by those where the predictive models do not estimate values in the constraints. Reducing the choice of AI accelerator for prototyping makes the design phase more sustainable, as fewer prototypes are needed, and consequently, fewer resources (e.g., metal, lithium) are used for them. Moreover, when offloading, the system designer can select the appropriated GPGPU considering the constraints and sustainable aspects like low power consumption. Based on the DFS analysis, designers are also enabled to configure the GPGPU in such ways as to consume less power, making the overall system more sustainable by avoiding performance losses.

Futhure Works

In further works, the proposed approaches will be generalized by considering different types of CNNs and increasing the technical setup platforms. Therefore, the following future works are planned:

- **Testing the capability for offloading strategies with Long Range Wide Area Network (LoRaWAN):** The mobile network types simulated in this thesis are characterized by high bandwidth, and the latest standard is also characterized by very low latency. Conversely, LoRaWAN is characterized by long-range with low bandwidth and high latency [132], consequently needs additional investigation on their ability for offloading of AI application. While performance improvement could only be detected by mobile networks with high bandwidth and low latency, power saving can already be seen in network types with low bandwidth. This indicates that LoRaWAN can also lead to power saving in offloading strategies. To increase the versatility of the decision support system from chapter 3, additional analysis on LoRaWAN has to be performed.

- **Development of a power meter system for external power measurements of the GPGPU:** As not all NVIDIA GPGPUs provide internal power meters and a recent study [131] pointed out that the internal power meter of NVIDIA GPGPUs has larger error than NVIDIA claims. A dedicated power meter system for the Peripheral Component Interconnect Express (PCIe) GPGPUs can lead to more accurate power measurements and thus improve the prediction further. This also enables benchmarking GPGPUs that do not offer an internal power metering with the nvidia-smi tool. Leading to an increase in samples for the training data set.

- **Extending analysis functionality of HyPA:** HyPA combines static and dynamic code analysis; hence, it is possible to combine metrics of both areas in the opensource implementation of HyPA. A current work-in-progress version has started to examine the possibilities and extend the number of static code analysis metrics like the McCabe metric (also called cyclomatic complexity). In future work, this version must be evaluated and published,

providing AI developer with more insights into the code implementation at compile time. Moreover, it is planned to integrate HyPA into a compiler like LLVM to increase user-friendly usage and integrate HyPA into the workflow of AI and software developer.

- **Increasing the variation of AI application to generalize the area of use of the predictive models:** With the increasing relevance of Large Language Models (LLMs) like Generative pre-trained transformers (GPT) (used in ChatGPT), the training data set must be extended by those NN used for LLMs. As the methods use general NN attributes that are available in all types of NNs, there is no need for modifications on the method. However, the models need retraining, and results can change with larger data sets, meaning other ML techniques can perform better.
- **Testing the applicability to another accelerator like CPU or FPGA:** All presented predictive models are limited to GPGPU as AI accelerator. While Zhang et al. [126, 129, 130] illustrates that similar approaches work on general applications for RISC-V, which suggests that combining the approaches to build a cross-platform predictive model, that can predict power consumption and performance for different types of accelerators like CPU, GPGPU or FPGA is possible. Such a cross-platform predictive model would enable the system designer to not only select the most appropriate GPGPU but the most appropriate AI accelerator out of a class of different accelerators.

Bibliography

[1] Milan Milenkovic. *Internet of Things: Concepts and System Design*. Springer, 2020.

[2] Thorsten Wuest, Daniel Weimer, Christopher Irgens, and Klaus-Dieter Thoben. Machine learning in manufacturing: advantages, challenges, and applications. *Production & Manufacturing Research*, 4(1):23–45, 2016.

[3] Ameeth Kanawaday and Aditya Sane. Machine learning for predictive maintenance of industrial machines using iot sensor data. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 87–90, 2017.

[4] Ahmed Zarrar, Muhammad Haseeb Rasool, Syed Mohammad Meesam Raza, and Ahmad Rasheed. Iot-enabled lean manufacturing: Use of iot as a support tool for lean manufacturing. In *2021 International Conference on Artificial Intelligence of Things (ICAIoT)*, pages 15–20, 2021.

[5] Christopher A. Metz, Mehran Goli, and Rolf Drechsler. Early Power Estimation of CUDA-Based CNNs on GPGPUs: Work-in-Progress. In *Proceedings of the 2021 International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '21, page 29–30, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Christopher A. Metz, Mehran Goli, and Rolf Drechsler. ML-based Power Estimation of Convolutional Neural Networks on GPGPUs. In *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 166–171, 2022.

[7] Christopher A. Metz, Mehran Goli, and Rolf Drechsler. Towards Neural Hardware Search: Power Estimation of CNNs for GPGPUs with Dynamic Frequency Scaling. In *Proceedings of the 2022 ACM/IEEE Workshop on Machine Learning for CAD*, MLCAD '22, page 103–109, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Christopher A. Metz, Christina Plump, Bernhard J. Berger, and Rolf Drechsler. HyBrid PTX Analysis for GPU accelerated CNNs inferencing aiding

Computer Architecture Design. In *2023 Forum on Specification & Design Languages (FDL)*, 2023.

[9] Christopher A. Metz, Mehran Goli, and Rolf Drechsler. Fast and Accurate: Machine Learning Techniques for Performance Estimation of CNNs for GPGPUs. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 754–760, 2023.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[11] M. Fingeroff. Machine learning at the edge: using hls to optimize power and performance. In *The Mentor - A Siemens Business (white paper)*, 2021.

[12] Nvidia. Nvidia v100 tensor core gpu. `https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf`, 2020. Accessed: 2022-11-02.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[15] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2017.

[16] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes. *CoRR*, abs/1711.04325, 2017.

[17] TOP 500. List statistics. `https://www.top500.org/statistics/list/`, 2022.

[18] Michael Feldman. New gpu-accelerated supercomputers change the balance of power on the top500. `https://tinyurl.com/2p8x74ww`, 2018.

[19] Fernanda Foertter. Summit gpu supercomputer enables smarter science. `https://developer.nvidia.com/blog/summit-gpu-supercomputer-enables-smarter-science/`, 2018. Accessed: 22.03.2022.

[20] Christopher A. Metz and Ralf Buschermoehle. Aufbau eines Machine Learning Clusters fuer Forschung und Lehre. In *20. DINI Jahrestagung*, 2019.

[21] Christopher A. Metz, Mehran Goli, and Rolf Drechsler. Pick the Right Edge Device: Towards Power and Performance Estimation of CUDA-based CNNs on GPGPUs. *CoRR*, abs/2102.02645, 2021.

[22] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. Gpu static modeling using ptx and deep structured learning. *IEEE Access*, 7:159150–159161, 2019.

[23] David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. The carbon footprint of machine learning training will plateau, then shrink. *Computer*, 55(7):18–28, 2022.

[24] Jie Tang, Dawei Sun, Shaoshan Liu, and Jean-Luc Gaudiot. Enabling deep learning on iot devices. *Computer*, 50(10):92–96, 2017.

[25] Alex de Vries. The growing energy footprint of artificial intelligence. *Joule*, 7(10):2191–2194, 2023.

[26] GPP. Strompreise privater haushalte in ausgewählten ländern weltweit im jahr 2022 (in us-dollar pro kilowattstunde) [graph]. https://de.statista.com/statistik/daten/studie/13020/umfrage/strompreise-in-ausgewaehlten-laendern/, 2023.

[27] Book Your Banner and Renewable Carbon. Impulspapier: Nachhaltigkeit im innovationssystem. 2020.

[28] EIA. Weltweiter Stromverbrauch in den Jahren 1980 bis 2021. https://de.statista.com/statistik/daten/studie/239764/umfrage/weltweiter-stromverbrauch/, 2023.

[29] BDEW. Nettostromverbrauch in Deutschland in den Jahren 1991 bis 2023 (in Terawattstunden) [Graph]. https://de.statista.com/statistik/daten/studie/164149/umfrage/netto-stromverbrauch-in-deutschland-seit-1999/, 2024.

[30] Christof Windeck and Christian Wölbert. Green it: Abwärme aus rechenzentrum für heizung nutzen. *c't Magazin*, 2022. `https://heise.de/-6327090`.

[31] Cloud and Heat. Whitepaper: Co2- und kosteneinsparpotenziale durch das cloud and heat-kühlsystem mit abwämenutzung in rechenzentren, 2023. https://www.cloudandheat.com/wp-content/uploads/2023/03/2019-12-16_Whitepaper-Einsparpotenzial.pdf.

[32] Yen-Lin Lee, Pei-Kuei Tsung, and Max Wu. Techology trend of edge ai. In *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–2, 2018.

[33] Seiji Mochizuki, Katsushige Matsubara, Keisuke Matsumoto, Chi Lan Phuong Nguyen, Tetsuya Shibayama, Kenichi Iwata, Katsuya Mizumoto, Takahiro Irita, Hirotaka Hara, and Toshihiro Hattori. A 197mw 70ms-latency full-hd 12-channel video-processing soc in 16nm cmos for in-vehicle information systems. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 100(12):2878–2887, 2017.

[34] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical power modeling of gpu kernels using performance counters. In *IGCC*, pages 115–122, 2010.

[35] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009.

[36] Thomas C Carroll and Prudence WH Wong. An improved abstract gpu model with data transfer. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 113–120. IEEE, 2017.

[37] Jianmin Chen, Bin Li, Ying Zhang, Lu Peng, and Jih-kwon Peir. Statistical gpu power analysis using tree-based methods. In *IGCC*, pages 1–6, 2011.

[38] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W. Cameron. A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In *SPDP*, pages 673–686, 2013.

[39] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. GPGPU performance and power estimation using machine learning. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576, 2015.

[40] Qiang Wang and Xiaowen Chu. Gpgpu power estimation with core and memory frequency scaling. *ACM SIGMETRICS Performance Evaluation Review*, 45(2):73–78, 2017.

[41] Lorenz Braun, Sotirios Nikas, Chen Song, Vincent Heuveline, and Holger Fröning. A simple model for portable and fast prediction of execution time and power consumption of gpu kernels. *ACM Transactions on Architecture and Code Optimization*, 18(1):1–25, 2021.

[42] Paola Busia, Svetlana Minakova, Todor Stefanov, Luigi Raffo, and Paolo Meloni. Aloha: A unified platform-aware evaluation method for cnns execution on heterogeneous systems at the edge. volume 9, pages 133289–133308, 2021.

[43] Qiang Wang and Xiaowen Chu. GPGPU performance estimation with core and memory frequency scaling. *IEEE IEEE Transactions on Parallel and Distributed Systems*, 31(12):2865–2881, 2020.

[44] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 725–737, 2015.

[45] Ioana Baldini, Stephen J. Fink, and Erik Altman. Predicting gpu performance from cpu runs using machine learning. In *SBAC-PAD*, page 254–261, 2014.

[46] Yehia Arafa, Abdel-Hameed A Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. Ppt-gpu: Scalable gpu performance modeling. *IEEE Computer Architecture Letters*, 18(1):55–58, 2019.

[47] Martin Lechner and Axel Jantsch. Blackthorn: latency estimation framework for cnns on embedded nvidia platforms. *IEEE Access*, 9:110074–110084, 2021.

[48] Rajesh Arumugam, Vikas Reddy Enti, Liu Bingbing, Wu Xiaojun, Krishnamoorthy Baskaran, Foong Foo Kong, A Senthil Kumar, Kang Dee Meng, and Goh Wai Kit. Davinci: A cloud computing framework for service robots. In *2010 IEEE international conference on robotics and automation*, pages 3084–3089. IEEE, 2010.

[49] Zicong Hong, Wuhui Chen, Huawei Huang, Song Guo, and Zibin Zheng. Multi-hop cooperative computation offloading for industrial iot–edge–cloud computing environments. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2759–2774, 2019.

[50] Muhammad Shiraz, Abdullah Gani, Azra Shamim, Suleman Khan, and Raja Wasim Ahmad. Energy efficient computational offloading framework for mobile cloud computing. *Journal of Grid Computing*, 13(1):1–18, 2015.

[51] Jianhang Tang, Guoquan Wu, Mohammad Mussadiq Jalalzai, Lin Wang, Bing Zhang, and Yi Zhou. Energy-optimal dnn model placement in uav-enabled edge computing networks. *Digital Communications and Networks*, 2023.

[52] Rolf Drechsler and Christopher A. Metz and Christina Plump. Energy-Efficient CNN inferencing on GPUs with Dynamic Frequency Scaling. In *Innovations in Data Analytics*, 2023.

[53] Christopher A. Metz and Rolf Drechsler. Challenges on Sustainable Artificial Intelligence Inference. *Communication of ACM*, 2024.

[54] Christopher A. Metz, Christina Plump, Bernhard J. Berger, and Rolf Drechsler. Performance and Energy Tradeoff Analysis of CNN Offloading to the Edge and Cloud Computing. In *2024 IEEE International Conference on Edge Computing and Communications (EDGE)*, 2024.

[55] Sana Hassan Iman, Christopher A. Metz, Lars Hornuf, and Rolf Drechsler. Classifying Crowdsourcing Platform Users' Engagement Behavior using Machine Learning and XAI. *In Mensch und Computer 2023 - Workshopband*, 2023.

[56] Christopher A. Metz. Machine Learning aided Computer Architecture Design for CNN Inferencing Systems. *arXiv preprint arXiv:2308.05364*, 2023.

[57] Sana Hassan Imam, Christopher A. Metz, and Rolf Drechsler. How Can Generative AI Curate the User Creativity on an Idea Crowdsourcing Platform? In *ACM CHI 24 Workshop on Generative AI in User-Generated Content*, 2024.

[58] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. Estimation of energy consumption in machine learning. *Journal of Parallel and Distributed Computing*, 134:75–88, 2019.

[59] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[60] Md Zahangir Alom, Tarek M Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S

Awwal, and Vijayan K Asari. The history began from alexnet: A comprehensive survey on deep learning approaches, 2018.

[61] Lukas Sekanina. Neural architecture search and hardware accelerator cosearch: A survey. *IEEE Access*, 9:151337–151362, 2021.

[62] Satya Mallick and Sunita Nayak. Number of parameters and tensor sizes in a convolutional neural network (CNN). https://learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network, 2018.

[63] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[64] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[65] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[66] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017.

[67] Mingxing Tan and Quoc V. Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, volume 97, pages 6105–6114, 2019.

[68] Tung D. Le, Gheorghe-Teodor Bercea, Tong Chen, Alexandre E. Eichenberger, Haruki Imai, Tian Jin, Kiyokuni Kawachiya, Yasushi Negishi, and Kevin O'Brien. Compiling ONNX neural network models using MLIR. *CoRR*, abs/2008.08272, 2020.

[69] Nvidia. Volta architecture whitepaper. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`. Accessed: 2022-01-18.

[70] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the nvidia ptx memory consistency model. In *Proceedings*

of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 257–270, New York, NY, USA, 2019. Association for Computing Machinery.

[71] NVIDIA. Kernel profiling guide - user manual. `https://docs.nvidia.com/nsight-compute/2020.1/pdf/ProfilingGuide.pdf`, 2020.

[72] Alvaro Saiz, Pablo Prieto, Pablo Abad, Jose Angel Gregorio, and Valentin Puente. Top-down performance profiling on nvidia's gpus. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 179–189, 2022.

[73] Yuan Lin and Vinod Grover. Using cuda warp-level primitives. `https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/`, 2018.

[74] Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. NVIDIA ampere architecture in-depth. `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`, 2020.

[75] Nvidia. CUDA toolkit documentation. `https://docs.nvidia.com/cuda/`, 2022.

[76] Nvidia. Parallel thread execution isa. `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html`.

[77] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. Flexible software profiling of gpu architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 185–197, New York, NY, USA, 2015. Association for Computing Machinery.

[78] Lorenz Braun and Holger Fröning. Cuda flux: A lightweight instruction profiler for cuda applications. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 73–81, 2019.

[79] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 3–12, 2009.

[80] Caroline Collange, Marc Daumas, David Defour, and David Parello. Barra: A parallel functional simulator for gpgpu. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 351–360, 2010.

[81] Jingyue Zhou, Yihuai Wang, Kaoru Ota, and Mianxiong Dong. Aaiot: Accelerating artificial intelligence in iot systems. *IEEE Wireless Communications Letters*, 8(3):825–828, 2019.

[82] Tuomo Sipola, Janne Alatalo, Tero Kokkonen, and Mika Rantonen. Artificial intelligence in the iot era: A review of edge ai hardware and software. In *2022 31st Conference of Open Innovations Association (FRUCT)*, pages 320–331, 2022.

[83] Shizhou Dong, Zhifan Gao, Sandeep Pirbhulal, Gui-Bin Bian, Heye Zhang, Wanqing Wu, and Shuo Li. Iot-based 3d convolution for video salient object detection. *Neural computing and applications*, 32:735–746, 2020.

[84] NVIDIA. Vergleich von nvidia jetson-modulspezifikationen.

[85] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors*, 20(9), 2020.

[86] Khadija Akherfi, Micheal Gerndt, and Hamid Harroud. Mobile cloud computing for computation offloading: Issues and challenges. *Applied Computing and Informatics*, 14(1):1–16, 2018.

[87] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.

[88] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. Association for Computing Machinery.

[89] Feixiang Li, Chao Fang, Mingzhe Liu, Ning Li, and Tian Sun. Intelligent computation offloading mechanism with content cache in mobile edge computing. *Electronics*, 12(5), 2023.

[90] Weijie Sun, Haixia Zhang, Leiyu Wang, Shuaishuai Guo, and Dongfeng Yuan. Profit maximization task offloading mechanism with d2d collaboration in mec networks. In *2019 11th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, 2019.

[91] B. L. WELCH. THE GENERALIZATION OF 'STUDENT'S' PROBLEM WHEN SEVERAL DIFFERENT POPULATION VARLANCES ARE INVOLVED. *Biometrika*, 34(1-2):28–35, 01 1947.

[92] F. E. Satterthwaite. An approximate distribution of estimates of variance components. *Biometrics Bulletin*, 2(6):110–114, 1946.

[93] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979.

[94] Daniel Christopher Hoinkiss, Jörn Huber, Christina Plump, Christoph Lüth, Rolf Drechsler, and Matthias Günther. Ai-driven and automated mri sequence optimization in scanner-independent mri sequences formulated by a domain-specific language. *Frontiers in Neuroimaging*, 2:1090054, 2023.

[95] Youcef Djenouri, Asma Belhadi, Gautam Srivastava, Uttam Ghosh, Pushpita Chatterjee, and Jerry Chun-Wei Lin. Fast and accurate deep learning framework for secure fault diagnosis in the industrial internet of things. *IEEE Internet of Things Journal*, 10(4):2802–2810, 2023.

[96] Sadegh Bafandeh Imandoust, Mohammad Bolandraftar, et al. Application of k-nearest neighbor (knn) approach for predicting economic events: Theoretical background. *International Journal of Engineering Research and Applications*, 3(5):605–610, 2013.

[97] Nvidia. Nvidia turing architecture whitepaper. `https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`. Accessed: 2022-01-18.

[98] Nvidia. Nvidia turing architecture whitepaper. `https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`. Accessed: 2022-01-18.

[99] Xinxin Mei, Qiang Wang, and Xiaowen Chu. A survey and measurement study of gpu dvfs on energy conservation. pages 89–100, 2017.

[100] Yuki Abe, Hiroshi Sasaki, Martin Peres, Koji Inoue, Kazuaki Murakami, and Shinpei Kato. Power and performance analysis of GPU-Accelerated systems. In *2012 Workshop on Power-Aware Computing and Systems (HotPower 12)*, Hollywood, CA, October 2012. USENIX Association.

[101] Paola Busia, Svetlana Minakova, Todor Stefanov, Luigi Raffo, and Paolo Meloni. Aloha: A unified platform-aware evaluation method for cnns execution on heterogeneous systems at the edge. *IEEE Access*, 9:133289–133308, 2021.

[102] Nvidia. nvidia-smi. `https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf`, 2016.

[103] Nvidia. Profiler user's guide. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`. Accessed: 2022-02-02.

[104] Ronny Kohavi and J. Ross Quinlan. *Data Mining Tasks and Methods: Classification: Decision-Tree Discovery*, pages 267–276. Oxford University Press, Inc., 2002.

[105] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[106] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[107] T. Patki and et al. Comparing GPU power and frequency capping: A case study with the MuMMI workflow. In *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 31–39, 2019.

[108] Johannes Doerfert, Atemn Patel, Joseph Huber, Shilei Tian, Jose M Monsalve Diaz, Barbara Chapman, and Giorgis Georgakoudis. Co-designing an openmp gpu runtime and optimizations for near-zero overhead execution. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 504–514, 2022.

[109] Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara Chapman. A virtual gpu as developer-friendly openmp offload target. In *50th International Conference on Parallel Processing Workshop*, ICPP Workshops '21, New York, NY, USA, 2021. Association for Computing Machinery.

[110] NVIDIA. Cupti documentation. `https://docs.nvidia.com/cupti/index.html`, 2022.

[111] Robert A. Bridges, Neena Imam, and Tiffany M. Mintz. Understanding gpu power: A survey of profiling, modeling, and simulation methods. *ACM Comput. Surv.*, 49(3), sep 2016.

[112] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009.

[113] gatech. GPU ocelot. `https://gpuocelot.gatech.edu/doxygen/`, 2012.

[114] David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[115] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[116] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.

[117] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks.

[118] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[119] Andrew W Appel. Ssa is functional programming. *Acm Sigplan Notices*, 33(4):17–20, 1998.

[120] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, 1989.

[121] Ali Haider Shamsan and Arman Rasool Faridi. Issues and challenges of using machine leaning in iot. In *2021 8th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 232–237, 2021.

[122] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[123] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1800–1807, 2017.

[124] Halima Bouzidi, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. Performance prediction for convolutional neural networks on edge gpus. In *ACM International Conference on Computing Frontiers*, page 54–62, 2021.

[125] Eugenio Gianniti, Li Zhang, and Danilo Ardagna. Performance prediction of gpu-based deep learning applications. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 167–170, 2018.

[126] Weiyan Zhang, Mehran Goli, and Rolf Drechsler. Early performance estimation of embedded software on risc-v processor using linear regression. In *25th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS). IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS-2022), April 6-8, Prague, Czech Republic*, 2022.

[127] Weiyan Zhang, Mehran Goli, Alireza Mahzoon, and Rolf Drechsler. Early performance estimation of embedded software on risc-v processors using ml algorithms. In *5th Workshop on RISC-V Activities. Workshop on RISC-V Activities, November 7-7, Berlin, Germany*, 2022.

[128] Weiyan Zhang, Mehran Goli, Alireza Mahzoon, and Rolf Drechsler. Ann-based performance estimation of embedded software for risc-v processors. In *33rd International Workshop on Rapid System Prototyping (RSP). International Symposium on Rapid System Protoyping (RSP-2022), October 13-14, virtuell*, 2022.

[129] Weiyan Zhang, Mehran Goli, Muhammad Hassan, and Rolf Drechsler. Efficient ml-based performance estimation approach across different microarchitectures for risc-v processors. In *Euromicro Conference Series on Digital System Design (DSD). Euromicro Conference on Digital System Design (DSD-2023), September 6-8, Durres, Albania*, 2023.

[130] Weiyan Zhang, Muhammad Hassan, and Rolf Drechsler. Towards ml-based performance estimation of embedded software: A risc-v case study. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV). ITG/GMM/GI-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation*

von Schaltungen und Systemen" (MBMV-2024), Kaiserslautern, Germany, 2024.

[131] Zeyu Yang, Karel Adamek, and Wesley Armour. Part-time power measurements: nvidia-smi's lack of attention, 2023.

[132] Jetmir Haxhibeqiri, Eli De Poorter, Ingrid Moerman, and Jeroen Hoebeke. A survey of lorawan for iot: From technology to application. *Sensors*, 18(11), 2018.