Faculty of Engineering Sciences
Heidelberg University

Master Thesis
in Computer Engineering
submitted by
Constantin Nicolai
born in Bretten, Germany
Day/Month/Year Here

YOUR TITLE HERE

This Master thesis has been carried out by Constantin Nicolai
at the
Institute of Computer Engineering
under the supervision of
Holger Fröning

ABSTRACT

Briefly summarize the contents of your work in 150-250 words. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

ZUSAMMENFASSUNG

Deutsche Version. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nice quote here.

— Some Author

ACKNOWLEDGMENTS

Your acknowledgments here if desired

CONTENTS

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Problem Statement 1
- 1.3 Scope 2
- 1.4 Contributions Overview 2
  - 1.4.1 Dataset Collection 2
  - 1.4.2 Prediction Model 2
  - 1.4.3 Validation 2

2 State of the Art and Related Works 5
- 2.1 State of the Art 5
- 2.2 Related Work 5
- 2.3 Research Gap 6

3 Dataset Collection 7
- 3.1 Operations 7
- 3.2 Time Profiling 7
  - 3.2.1 Inference 9
  - 3.2.2 Training 9
  - 3.2.3 Proportionality 10
- 3.3 Energy Profiling 10
- 3.4 Log file Evaluation 11
- 3.5 GPU Clocks 12

4 Prediction 13
- 4.1 Model Selection 13
- 4.2 Prediction Architecture 13
- 4.3 Prediction Workflow 15

5 Validation 17
- 5.1 Dataset Validation 17
  - 5.1.1 Methodology 17
  - 5.1.2 Hardware Platforms 17
  - 5.1.3 GPU Configurations 18
  - 5.1.4 Base Results 18
  - 5.1.5 Tensor Core Real-World Impact 22
- 5.2 Prediction Accuracy 22
  - 5.2.1 Operations Level 22
  - 5.2.2 Neural Network Level 24

6 Discussion and Outlook 31

A Appendix 33

1 INTRODUCTION

1.1 MOTIVATION

The global increase in usage of machine learning applications illustrates an acceleration in adoption across both industry and the private sector. The unfathomably large energy costs tied to this broader adoption have already prompted a change in public sentiment towards energy energy infrastructure. Plans for building trillion-dollar data centers are emerging, necessitating the re-commissioning of previously decommissioned nuclear power plants, which were originally phased out as part of nuclear energy reduction efforts. This reversal of nuclear phase-out policies underscores the significant infrastructural and political pressures exerted by the energy requirements of machine learning technologies.
In this landscape it is more pressing than ever to gain insight into the roots of the energy costs in order to optimize future developments on an informed basis.
In order to facilitate a more informed pairing of workload and GPU we introduce a framework to help guide the decision towards an optimal choice. This way regardless whether the fastest execution or the smallest energy footprint is desired, the informed choice enabled by our framework prevents wasteful computation.

1.2 PROBLEM STATEMENT

While a considerable amount of previous work has been done in profiling and prediction of neural network performance, no prior work covers the same cases and performance metrics targeted in this work. Therefore, our study investigates both training and inference cases covering both execution time and power consumption.
This contribution is valuable because in most cases where a new model architecture is designed or an existing architecture is adapted, both the training and the inference efficiency are relevant at some point of the models lifespan. At the same time, latency or power envelope requirements may be fluent between the training and inference stages, necessitating the study of both performance metrics.

1.3 SCOPE

This can only be written, when research is finished.

1.4 CONTRIBUTIONS OVERVIEW

1.4.1 Dataset Collection

In our first contribution we introduce our method for collecting profiling data and creating a coherent dataset by processing the collected data.
The profiling measurement is executed via a python script, utilizing the Pytorch Benchmark library[1] to conduct the execution time measurements. The power measurements are conducted using the command-line tool nvidia-smi[2] being run in the background while the benchmark is performed.
In order to enable repeatability the execution times and power readings, along with their respective measurement errors are stored together with the Pytorch objects for profiled operations, along with a sensible selection of related metrics.

1.4.2 Prediction Model

In our second contribution we present our prediction model. Based on the predictions for the operations occurring in a neural network we can infer a prediction for the complete network. We can therefore provide insight into which of the studied GPUs is most suitable for a given neural network, depending on the metric of interest. This way we can identify both the GPU which can execute the neural network in the fastest time, and the GPU which results in the smallest energy consumption and accompanying heat output.

1.4.3 Validation

In our third and final contribution we investigate the quality of our dataset of measurements for individual operations by comparing the sum of individual operations against a full neural network run for both execution time and energy consumption.
In the second part of this contribution we assess the accuracy of our predictions both for individual operations as well as for full neural networks. Both for the individual operations as well as for the full neural networks this is achieved by comparing the predictions to

1 https://pytorch.org/tutorials/recipes/recipes/benchmark.html
2 https://docs.nvidia.com/deploy/nvidia-smi/index.html

measurements acquired using the same measurement methodology used to collect the initial dataset.

2 STATE OF THE ART AND RELATED WORKS

2.1 STATE OF THE ART

The challenge of predicting neural network performance has invited a plurality of approaches. Apart from the methodological approaches they also differ in a number of aspects. While execution time is commonly the metric of choice, only few go further and also study metrics like power consumption and memory footprint. Another important distinction is the workload studied in the work, more specifically, whether both training and inference are studied. For practical reasons it is also relevant which machine learning framework is used and what hardware targets are required and can be predicted for. These many dimensions of possibility result in no work covering all possibilities, but allows for many different approaches which have use cases in a given situation.

2.2 RELATED WORK

Kaufmann et al. take an approach of performance modeling by means of the computation graph. They are however limited to the Google Tensor Processing Unit in this work.
Justus et al. take an approach exploiting the modular and repetitive nature of DNNs. Given the same operations are repeated over and over in training, often only varying in a few key paramters, these execution time for these base building blocks is measured. This is then done for one batch in the training process and generalized to the whole training process from there. There is however no presentation of the methodology for the execution time measurements.
Qi et al. present PALEO which employs an analytical approach towards predicting the execution for both training and inference of deep neural networks. The analytical approach brings both advantages and disadvantages with it. It does not require a dataset of measured execution times as a training set in the same way many other works do, but on the other hand it also is based on more fixed assumptions about the DNN execution than a more data driven approach.
Wang et al. approach with a mult-layer regression model to predict execution time for training and inference. Their work is however rather limited in terms of hardware targets and different DNNs studied.
Cai et al. focus their work, NeuralPower, on CNNs running on GPUs. For each target GPU, they collect a dataset and fit a sparse polynomial

regression model to predict power, runtime, and energy consumption. While NeuralPower achieves good results, its usefulness has become limited due to its exclusive focus on CNNs, as other DNN architectures have grown in popularity.
Gianitti et al. also exploit the modular nature of DNNs in their approach. They define a complexity metric for each layer type, optionally including backpropagation terms, allowing them to predict execution times for both training and inference. However, their method faces significant limitations, as the complexity metric is only defined for a specific set of operations, making it incompatible with networks that include layers not covered in the original work. As a result, their approach is essentially limited to classic CNN architectures.
Velasco-Montero et el. also take the familiar per-layer approach. Their predictions are based on linear regression models per type of layer, but again for a specific set of predefined operations. Given their focus on low-cost vision devices these restrictions are reasonable, but limit generalizability.
Sponner et al. take a broad approach in their work. It works in the TVM framework giving it high flexibility in target hardware and studied metrics. It is in fact the only work to include execution time, power consumption and memory allocation. Given the automated data colletion used to create the dataset basis for the predictions, there are also few limitations to the networks that can be studied with this. The predictions are based on an extremely randomized tree (ERT) approach with XGBoosting applied. The only major drawback for this work is its limitation to only study inference, due to TVMs limitation to inference.

2.3 RESEARCH GAP

With all these very different works no single one was able to cover all possible angles to interest, although Sponner et al. got rather close. But given the current landscape of available publications our work will focus on finding the best GPU for a PyTorch job. In order to achieve that, we will cover execution time, power and energy consumption and will provide inference and training predictions for these metrics. Our approach also employs a different method of automatic dataset collection, which allows for a broad field of study. In order to obtain power readings are collected directly through nvidia-smi. While due to the scope of this work this limits us to Nvidia GPUs, the methodology could just as well be applied to any other hardware target which supports reporting power readings.

3 DATASET COLLECTION

This contribution outlines the method used to collect a profiling dataset. The dataset serves as a training set for the prediction model. The parameters covered are various Torchvision models and input sizes, execution time and power, both the inference and the training case as well as different GPUs and various GPU clocks.

3.1 OPERATIONS

In order to increase generality our approach exploits the layerwise structure of DNNs. This is achieved by working on the layer level rather than on the model level. In order to be rigorous we need to be clear on the terminology. The workload and its characteristics depend on the layer and its input features, just like it would on the DNN and the input dimensions on the model level. We are interested in most general objects with an identical characteristic workload. On the layer level those will be layers with specific input feature dimensions and layer settings. We will refer to these objects as operations. On the model level those will be DNNs with specific input image dimensions, which we will refer to as model-input sets.
The units for which we will make predictions later are individual operations. To this end, we aim to collect a dataset of operations and profile the execution time and wattage for each one. To ensure that the dataset reflects operations encountered in real-world scenarios, we select a number of representative model-input sets from models of the Torchvision library. The set of operations which we will be profling, is the set of all operations occuring in any of these model-input sets.
Extracting this set is automated via scripts, but can just as well be done by hand if so desired. With this set at hand we can dive into the actual profiling.

3.2 TIME PROFILING

The time profiling is performed using the benchmark utility from `torch.utils.benchmark` called Timer. This enables us to run our benchmark function with the specific layer and input features as input parameters, allowing us to run it for each operation.
To achieve more stable results and inprove the simultaneous power profiling it is desirable to run the benchmark for each operations

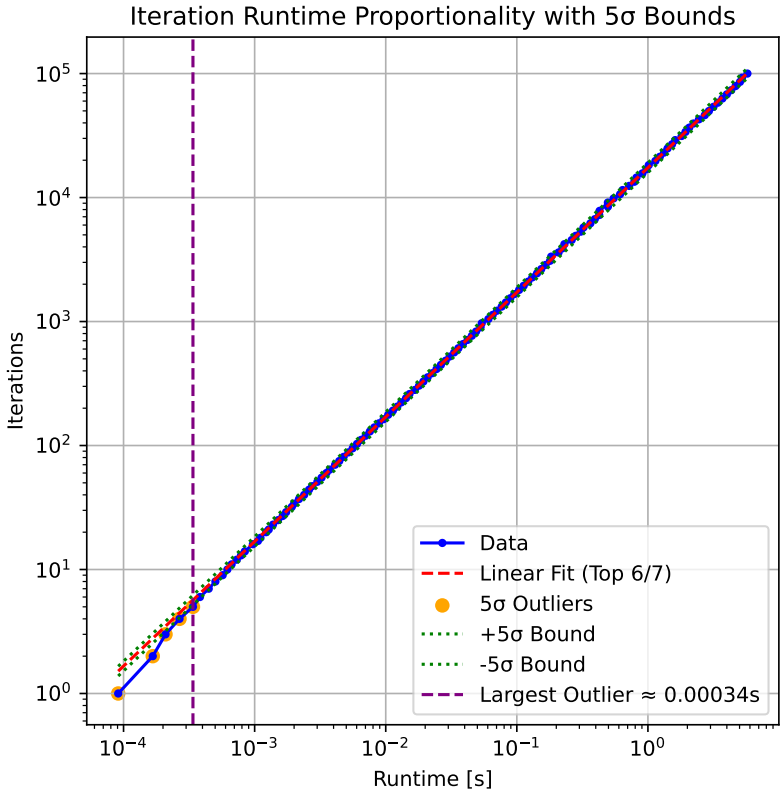


Figure 3.1: This plot shows the proportionality of the runtime with the number of iterations ran for an operation. A linear fit is performed to the $\frac{6}{7}$ of data points with most iterations. From this fit the standard deviation for the linear section is found. This allows us to identify the largest outlier below which the proportionality breaks. In order to do this we search for the largest runtime in the 5σ outliers.

for at least a few seconds. This is achieved by setting a minimum runtime for the benchmark to aggregate statistics. Different workload characteristics and different computational capabilities depending on the GPU configuration being profiled result in different requirements for individual profiling runs in order to ensure sufficient repitions in the measurement process. In order to accomodate this requirement, the runtime of the measurement is configurable via a command-line parameter.
For the initialization of the operations we have to be careful to avoid two extreme cases in order to achieve a sensible approximation of the execution characteristics within a neural network. The first case we want to avoid is initializing one instance of the operation and performing each benchmark loop on that instance. This would result in hitting the cash every time and would not be representative. The second extreme we want to avoid is initializing a new instance for each loop of the benchmark run, which adds the initializition overhead and the larger latency of having to access the GPU main memory every time, skewing our results in the opposite direction. We have therefore chosen the middleground approach of initializing a sizable block of operation instances with random weights and biases, which is then looped over by the benchmark kernel, resulting in some reuse of layers, stressing the memory system in a sensible manner.

3.2.1 Inference

The central difference between inference and training profiling appears in the design of the benchmark function.
For inference profiling we put our operations into `eval` mode and call the operations within a `torch.inference_mode` environment, which is equivalent to the exection in a typical inference forward pass through a model. Within the loop we call the operator on a preinitialized random input tensor for the forward pass. The input tensor is a single instance, as it represents the activations from the previous layer which can be expected to live in high level memory in our memory hierarchy. Since this forward pass is everything we want to profile for inference, this benchmark loop is sufficient.

3.2.2 Training

For the training profiling we put our operations into `train` mode and omit the environment used in the inference case. In order to portray the training process for a single operation, we need to run a forward pass and a backward pass. The forward pass is handled identically. In order to execute the backward pass we need a substitute for the gradients flowing backward trough the model. This is achieved by preinitializing

a random torch vector with the dimensions of our operation's output which we can then call the backward pass on. Other than that, we only have to make sure the runtime keeps the gradients for all operation instances and keep resetting the input vector gradients for each loop iteration.

3.2.3 Proportionality

Since we are assuming the number of benchmark iterations and the resulting total runtime to be proportional, we need to test that assumption. In order to do that we plot their relationship in Figure 3.1. By identifying how short the total runtime needs to become for the linear relationship to break, we can find a lower bound to our desired benchmark duration for each operation we study. The result of this investigation is that as long as we stay above 100 ms we are on the safe side. In our actual measurements every operation is benchmarked for at least several seconds, even though the exact duration may vary depending on the computational intensity of the specific operation.

3.3 ENERGY PROFILING

The energy profiling is conducted by measuring the power in watts. Combined with the time measurements this gives us the energy results.
We are starting `nvidia-smi` as a background process and logging the power readings into a temporary csv file, which is evaluated later in the script to extract the statistics relevant to our purpose. On a higher level abstraction, a second instance of `nvidia-smi` runs during the complete benchmark process. This log can be compared to the timestamps included in the dataset of profiled operations, in order to investigate surprising annomalies in the results. However, the results relevent to our power profiling are calculated instanly from the temporary csv log file. Due to the nature of our meausrement pipeline, some preparations and processing after the fact are necessary to ensure robust results.
As can be seen in Figure **??**, showing the power measurement over time for alternating idle times and benchmark calls, the transition is not instant. There are some power readings in between the steady states of idle and benchmark.
In order to illustrate the existance of a startup effect without idle times between the benchmark runs, Figure **??** shows five looped benchmark runs of the same benchmark overlayed. For each run we can observe the startup effect.
We do not want to keep this startup effect in our result, because it is a result of our benchmark execution and not representative of the much

more integrated execution pipeline in a neural network.
In order to keep it from affecting our results, we use a 3σ channel around the initial mean of the power readings and drop everything else.
As a second precaution, the script also employs some warmup runs in order to further mimimize the impact of the startup effect.

3.4 LOG FILE EVALUATION

The following section explains the details of how we arrive at our results from log files collected in the benchmark runs. Below you can see a snippet from one of the log files. The fourth entry in each row is the power reading.

```
2024/10/10 13:18:58.369, 81, 145.99, 4396, 11264
2024/10/10 13:18:58.407, 81, 182.11, 4396, 11264
2024/10/10 13:18:58.428, 81, 182.11, 4396, 11264
2024/10/10 13:18:58.439, 81, 182.11, 4396, 11264
2024/10/10 13:18:58.490, 81, 178.50, 4396, 11264
2024/10/10 13:18:58.514, 81, 178.50, 4396, 11264
2024/10/10 13:18:58.538, 81, 178.50, 4396, 11264
```

Let us begin with the power evaluation. The log file is read in via `pandas`[1]. Any existing rows containing a non-numerical value are dropped from the dataframe. We then find the standard deviation for the power and drop all rows containing a power reading outside a $3\,\sigma$ range. The following formulae are used:

$$\overline{W} = \frac{1}{n}\sum_{i=1}^{n} W_i \tag{3.1}$$

$$\sigma_W = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(W_i - \overline{W})^2} \tag{3.2}$$

$$W_{filtered} = W \text{ such that } |W - \overline{W}| < 3\sigma_W \tag{3.3}$$

With n being the number of timestamps and W being the power.
The same two formulae are used to find the mean power $\overline{W}_{filtered}$ and standard deviation $\sigma_{\overline{W}_{filtered}}$ of the filtered power.
Both the total runtime t_{tot} and its standard deviation $\sigma_{t_{tot}}$ are provided by `torch.utils.benchmark`. With power and time evaluated, we find the total run energy E_{tot} and its error $\sigma_{E_{tot}}$.

1 pandas

$$E_{tot} = \overline{W}_{filtered} \cdot t_{tot} \tag{3.4}$$

$$\sigma_{E_{tot}} = \sigma_{\overline{W}_{filtered}} \cdot t_{tot} \tag{3.5}$$

From this, we find the time per iteration t and the energy per iteration e, as well as the error for the time per iteration σ_t and for the energy per iteration σ_e with the number of iterations being N.

$$t = \frac{t_{tot}}{N} \tag{3.6}$$

$$e = \frac{E_{tot}}{N} \tag{3.7}$$

$$\sigma_t = \frac{\sigma_{t_{tot}}}{N} \tag{3.8}$$

$$\sigma_e = \frac{\sigma_{E_{tot}}}{N} \tag{3.9}$$

The most interesting results here are of course the time and enery per iteration with their respective errors, as well as the average power, all specific to the tested opeation of course.

3.5 GPU CLOCKS

The last parameter we needed to build a test methodology for is the GPU clock. More precisely the core clock of the GPU. Because using `nvidia-smi` to change the clock requires lower level permissions, this was done in a seperate script with different permissions, which was then called from the `sbatch` script used to run the benchmark. This clocking script takes command line parameters to set the desired clock speed. A range of clock speeds was tested in order to gain insight into the relationship between clock speed, runtime and energy consumption. These clock speeds were simply set manually in the `sbatch` script.

4 PREDICTION

In this chapter we will introduce our model for providing predictions based on the collected dataset. We will go into the idea and decisions in creating it in this specific way and provide insight into the implementation.

4.1 MODEL SELECTION

Since we are interested in time and energy, we need two prediction models. More precisely, we need one for runtime predictions and one for power predictions. Multiplying the two predictions we obtain the our predicted energy.
In order to retain as much understanding of the prediction process as we can while ensuring tolerable execution times for the prediction times, we do not want to use a full DNN to perform the predictions. Instead we are looking for a more lightweight solution, closer to classical statistics. These requirements led us to using a random forest predictor model. It is both lightweight and sets restrictions to its input which force us to format our dataset in a way that provides some insight into cause and effect for the predictions.
From the same family of tree based ensemble methods, we also ran some tests with Extremely Randomized Trees, but the increased randomness did not yield better results. We therefore continued with the less complex random forest. A similar attempt was made with Extreme Gradient Boosting, but this also did not provide and increase in prediction accuracy justifying the increased training time. While we did not observe any reason to use on of these methods, we do not want to dissuade others from doing so. Switching around between the methods is relatively simple, due to the provided implementations from sci-py, and the fact, that we did not find improvements using them on our dataset does not necessarily have to mean there are no benefits when studying a different dataset.

4.2 PREDICTION ARCHITECTURE

As we are using the sci-py implementation of the random forest model, we did not have to create architecture. Instead most work went into formatting and preprocessing the dataset. In order to give the model as much useful input information as possible we needed to provide

the operator name, the input size and the GPU clock along with all potentially useful attributes of the operation's pytorch object to the predictor.
Because random forest model only takes numerical inputs, we make sure we format the input vector in a suitable way. For the operator names, this means we have a fixed number of categories in our dataset and can therefore use one-hot encoding to identify each type in a way that is readable to the random forest model. Another challenge arises from the fact, that different operations do not always have the same attributes. Along the same lines, the length of the input size tuple also is not equal for all operators. In order to preserve generality, we want to support all operators in one random forest model, which in turn means, we need to find a solution for this asymmetry in attributes for the different operators. The approach we ended up using, was to introduce a Boolean flag for each attribute entry in the input vector, signifying whether is applies for this operator.
For example, a linear layer expects an input tensor of the shape: `(batch_size, in_features)`, but a `Conv2d` layer expects: `(batch_size, in_channels, height, width)`. With that, the input size tuple for for a linear layer with `(batch_size=32, input_features=128)` would be encoded as `(32,1,128,1,-1,0,-1,0)`, whereas the input size tuple for a `Conv2d` layer with `(batch_size=32, in_channels=16, height=256, width=256)` would result in `(32,1,16,1,256,1,256,1)`. This way we can construct a meaningful input vector for the random forest model, which has a constant length and meaningful entries. We use `-1` as our entry for non applying fields, as there are no negative input sizes.
In a similar fashion, there is a field in the input vector for `stride`. For convolutions, this carries the information of the stride, but for other operators like linear layers or ReLUs, it simply carries a minus one, with a zero flag in the next entry signifying it does not apply for this operator.
This encoding approach is used on predefined list of attributes which are likely to have an impact on the computational characteristics of the operation. The choice of these parameters was conducted in a heuristic fashion, based on our understanding of the operators in question. However, further work could try to minimize the number of necessary attributes by quantitatively investigating their computational impact and including or excluding them accordingly. With the heuristic approach used here, we have attempted to err on the side of rather including too many attributes then to few.
The last important input vector entry we need to mention is the GPU clock speed. Even though the initial implementation trained runtime and power random forest predictors for each individual clock speed in our dataset, treating the GPU clock as a parameter for the model reduces unnecessary complexity without having a measurably negative impact on the prediction accuracy. This way it is just another parame-

ter for the predictor, simplifying both the training of the random forest models, as well as their utilization, since it is no longer necessary to locate the specific model for the GPU clock speed of interest, and can instead just be fed into the predictor.

4.3 PREDICTION WORKFLOW

As apparent from the architecture, our prediction model operates on the operations level. Is this is what is desired for a given study, that is great. One can simply provide the operations of interest with their respective input size and GPU clock to preprocessing and prediction script and will receive predictions.
Nevertheless, we expect most users to want to make predictions for whole neural networks. In that case the workflow is very similar to the dataset creation. It starts with extracting the operations, and how often they occur. Afterwards all unique operations will be ran through the predictor and the results summed up according to how often they occur in the model.
Critically, due to the modular nature of the approach, there is no need to the neural network being predicted to be actually executed or even exist in a finished form. As long as the operations are available it and it is known how often they occur predictions can be made.

5 VALIDATION

While the previous contributions provided insight into the building blocks of this work, this chapter will serve to show the results it can provide. By providing quantitative results we can validate the methodology and provide an informed impression of its capabilities and limitations.

5.1 DATASET VALIDATION

In this first section our focus is on ensuring the datasets we collect for training are reasonable accurate. Otherwise we would base our predictions on a training set which might be completely of the mark, defeating the purpose of the predictions from the very start.

5.1.1 Methodology

Our approach to validating the datasets is rather simple in principle. Our measurements for the individual operations are supposed to be representative of their execution withing a DNN execution. So in order to verify this is the case, we compare measurements of runs of the complete DNNs, to summed up result from our individually measured operations.
The measurements for the complete DNNs are conducted with the same measurement pipeline used to measure the individual operations, since to the script a DNN can simply be viewed as just one larger operation. We built the script which extracts the unique operations present in a DNN in a way that is also tracks how often each unique operation occurs. Because of that, we can use that information to simply sum the results from our collected dataset accordingly.

5.1.2 Hardware Platforms

We begin by looking into our findings for the three original GPU configurations. Later we will also look into further results with different clock speeds.
The two hardware platforms studied here are the Nvidia RTX 2080 TI and the Nvidia A30. The Nvidia RTX 2080 TI is based upon the Turing architecture from the year 2018 and features 4352 CUDA cores and 544 first generation tensor cores with FP16 support. The Nvidia A30 is

based upon the Ampere architecture form the year 2020 and features 3584 CUDA cores and 224 second generation tensor cores with TF32 support.

5.1.3 GPU Configurations

Given the capability of floating point 32 computation on the A30's tensor cores, we decided to have a look its different performance between having its tensor cores enabled or disabled. This differentiation was not feasables for the 2080TI. We use `FP32` as the data type in our benchmarks, and its tensor cores do not support that.
This resulted in three GPU configurations for this section. The 2080TI with default settings, the A30 with default settings and the A30 with its tensor cores manually disabled.

5.1.4 Base Results

5.1.4.1 RTX 2080 TI

Looking at the results for the 2080TI our findings are not perfect. The agreement between measured and summed results looks rather promising for larger model-input sets. However, for smaller ones, there are instances where the agreement between measurement and summation is less than ideal. The model-input sets displaying this behavior are the EfficientNetB0 (32, 3, 224, 224), the ResNet18 (32, 3, 32,32) and the ResNet34 (32, 3, 56, 56). In these instances, the summation method overestimates both runtime and energy. However, the overestimation is more pronounced for runtime than for energy.

5.1.4.2 A30 Tensor Cores Disabled

Our findings for the A30 with its tensor cores disabled are already more consistent than 2080TI's. While the same trends are visible, they are much less pronounced and our summation approach yields closer approximations of the measured results overall.

5.1.4.3 A30 with Tensor Cores Enabled

Our findings for the A30 with its tensor cores enabled are also very consistent. While the trends we observed for the 2080TI are not completely gone, they are even less pronounced then for the A30 with disabled tensor cores, giving us the best results of the three hardware configurations.

5.1.4.4 Runtime Errors

For all runtime results, the standard deviation is very small, both for the summation and for the measured runtimes. Given that, it is clear that the discrepancies between the two cannot be solely caused by statistical noise.
Unfortunately, since this dataset validation is not they primary focus of this work, a deeper dive into its error estimation falls outside the scope.

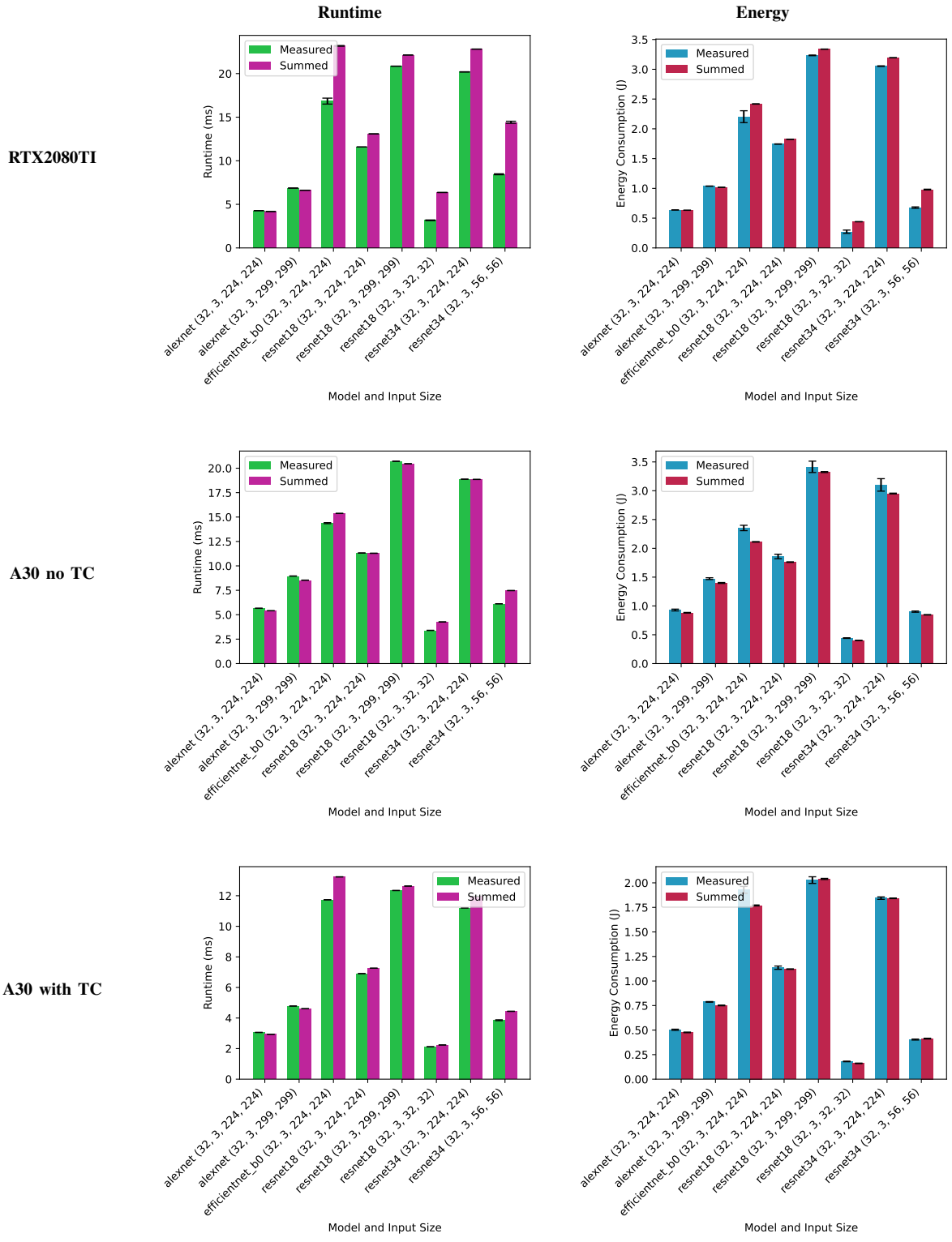


TABLE I: These are the plots for smaller model-input sets. The ones in the left column show the runtimes we measured with full model-input runs compared to our approximations. The ones in the right column show the same comparison for the energy consumption. The approximations were obtained by summing the individual findings for all operations within the model-input set. The error bars show the standard deviation, for the summed approximation the result of error propagation of the individual standard deviations.

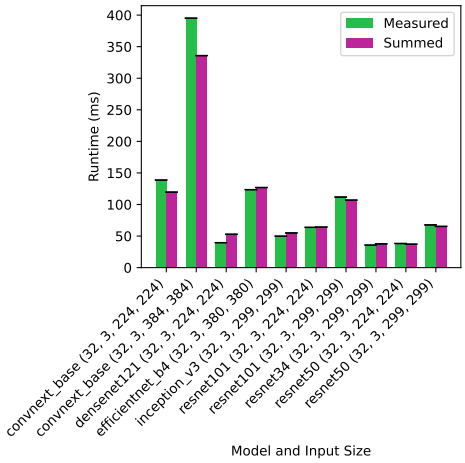
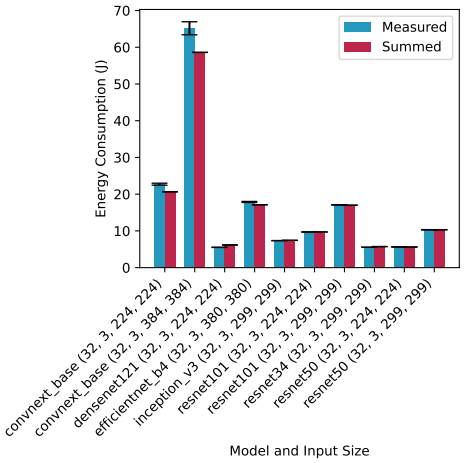
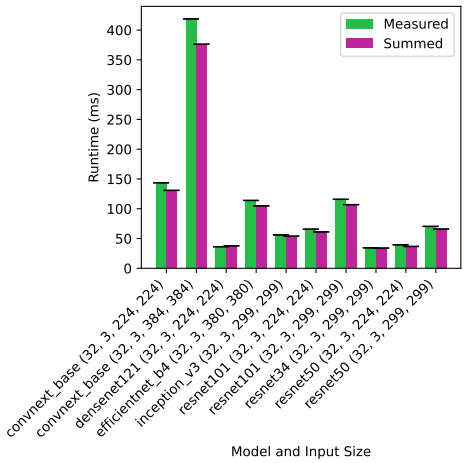
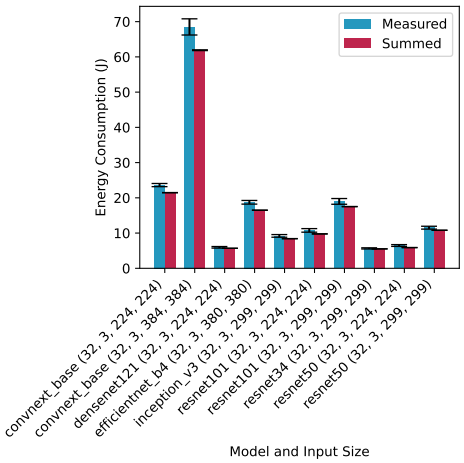
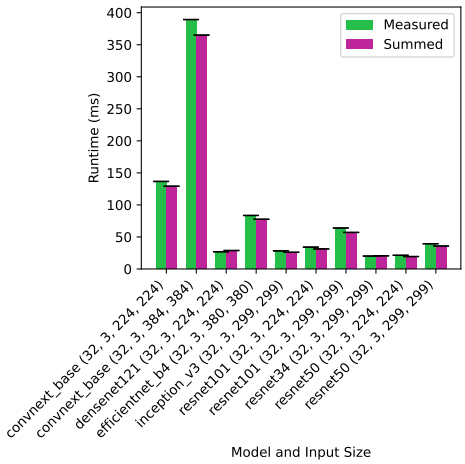
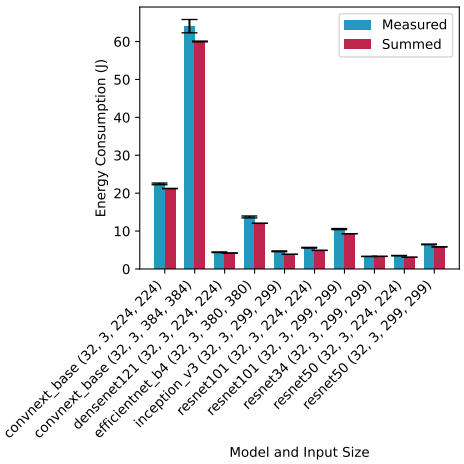
	Runtime	Energy
RTX2080TI		
A30 no TC		
A30 with TC		

TABLE II: These are the plots for larger model-input sets. The ones in the left column show the runtimes we measured with full model-input runs compared to our approximations. The ones in the right column show the same comparison for the energy consumption. The approximations were obtained by summing the individual findings for all operations within the model-input set. The error bars show the standard deviation, for the summed approximation the result of error propagation of the individual standard deviations.

5.1.5 Tensor Core Real-World Impact

As can be seen in Figure 5.1 and Figure 5.2 showing the measured energy for the full model-input set runs on all three GPU configurations, tensor cores do have a significant impact on the energy efficiency of running PyTorch models.
This difference is more pronounced for smaller model-input sets and appears to become continually smaller for larger and more complex ones. But the difference does not appear to simply be proportional to the model's energy cost either. At first glance and without studying the individual model architectures in detail, it would appear that the difference decreases with the model's dependency complexity.
Dependency complexity is used here to describe both the amount and the depth of dependencies, measured by the number of layers they span, when dependencies go beyond direct, sequential connections between adjacent layers.
When comparing the results for different flavors of ResNets to the results for model architectures with higher dependency complexity such as ConvNext, EfficientNet and DensetNet, it can be seen that the results are much closer for the latter ones, while for the ResNets the tensor cores get to show their potential.
Taking a step back from studying the impact of the tensor cores, there are also interesting findings in comparing the results for the 2080TI to the other GPU configurations. We find worse energy efficiency for the 2080TI compared to the A30 running with tensor cores for all models. But when the tensor cores are disabled this trend gets reversed. Overall the difference between the 2080TI and the A30 without tensor cores is smaller than the difference between the 2080TI and the A30 with its tensor cores enabled. However, the pattern of the energy efficiency being best on the A30 with tensor cores, the 2080TI occupying the middle position, and the A30 without tensor cores having the worst energy efficiency remains the same for all model-input sets.

5.2 PREDICTION ACCURACY

With this section we are moving on from results and insight gained directly from the dataset collection and move into our findings for the prediction model.

5.2.1 Operations Level

Because the prediction model works on the operations level, we will begin by evaluating it on the operations level.
We are using the implementations from the `sklearn` library and have investigated XGBRegressor, ExtraTreesRegressor and RandomFore-

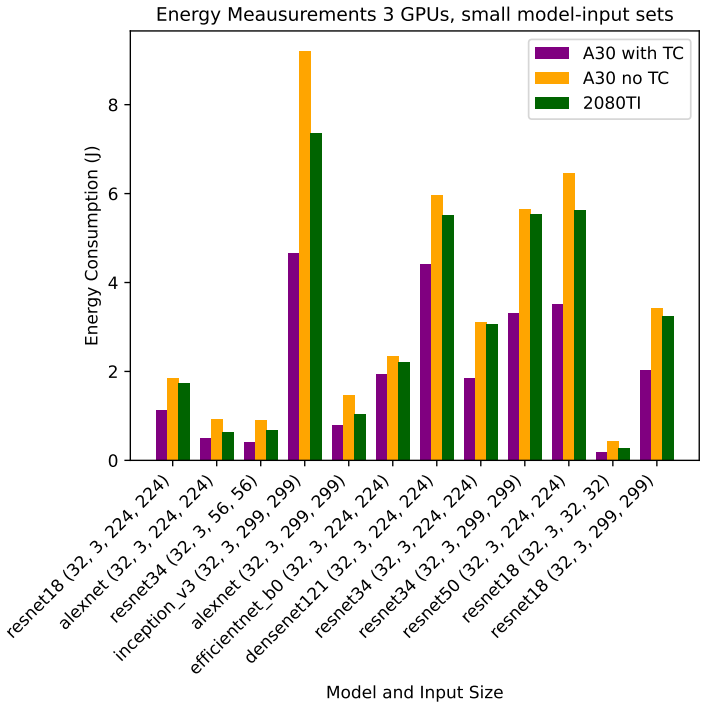


Figure 5.1: Comparison of energy measurements for the 2080TI and the A30 with tensor cores once disabled and once enabled. The resulting ordering is identical for all model-input sets. However, the relative differences show a lot of variation, being more pronounced for these smaller model-input sets.

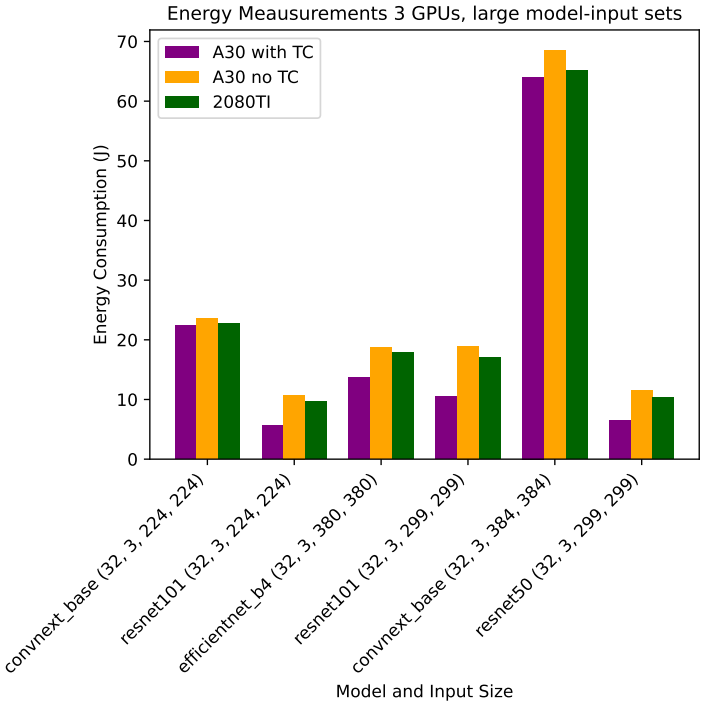


Figure 5.2: For these larger model-input sets, we maintain the ordering, but we observe far weaker relative differences between the hardware configurations. For model-input sets with a high dependency complexity, like the ConvNext Base model, we find the most similar energy results across the configurations.

stRegressor to use as our prediction model. Initial test showed clearly that `ExtraTreesRegressor` performed worse than the others in terms of predictive power, so we focused on the other two.
We chose to use the coefficient of determination, also known as R^2, as our metric for the prediction accuracy. A 15-fold cross validation was run and resulted in mean R^2 for the runtime model over all cross validation runs of $\overline{R^2}_{time} = 0.876 \pm 0.073$. For the power model we found a mean R^2 of $\overline{R^2}_{power} = 0.977 \pm 0.007$.

5.2.2 Neural Network Level

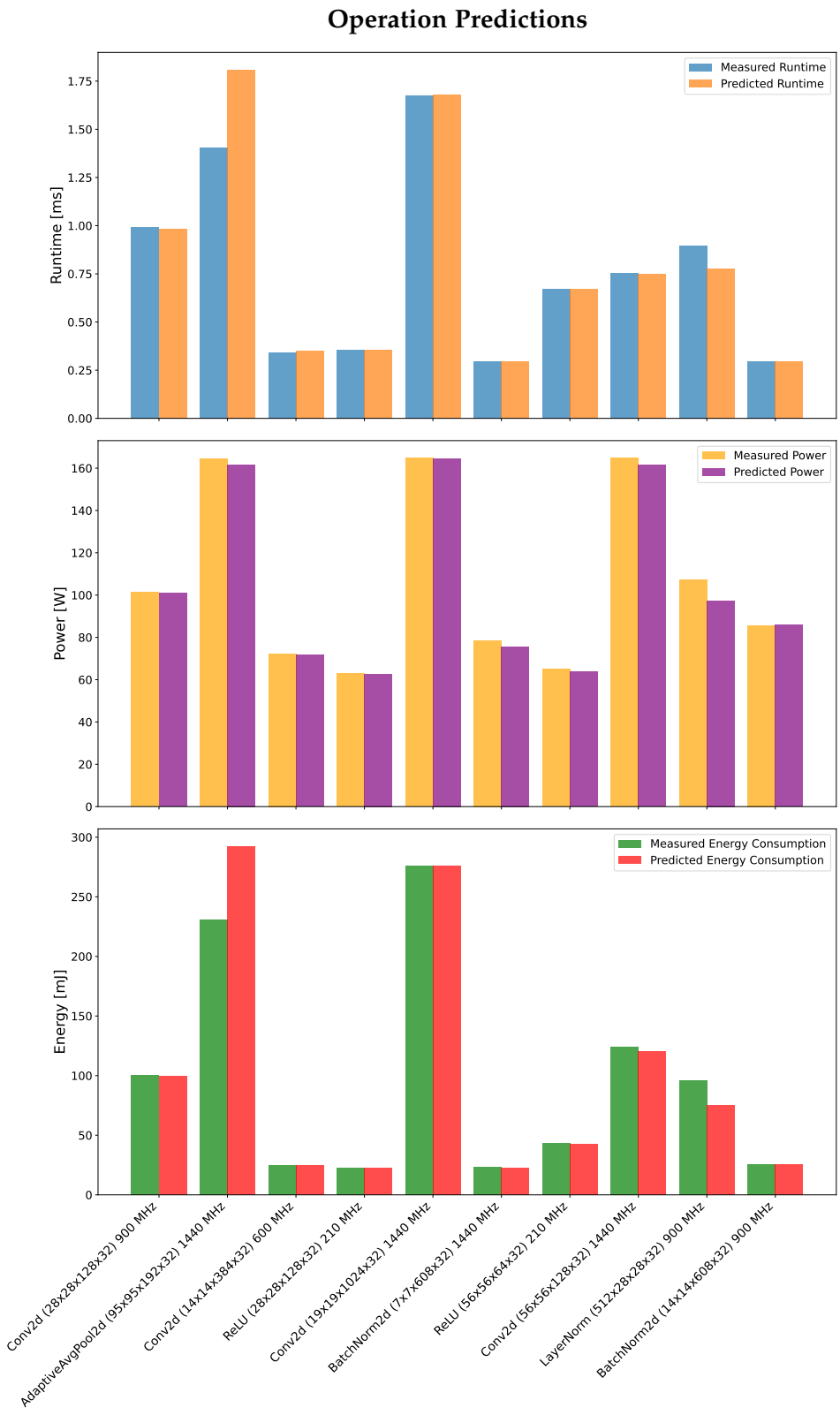


Figure 5.3: Comparison of predictions to measurements for ten operations from the test set. While we have outstanding agreement for most of them, we can see the limits of our predictive power in the results for operations 2 and 9. This illustration can give us a more intuitive impression of the R^2-error of 0.92 for the runtime and of 0.97 for the wattage.

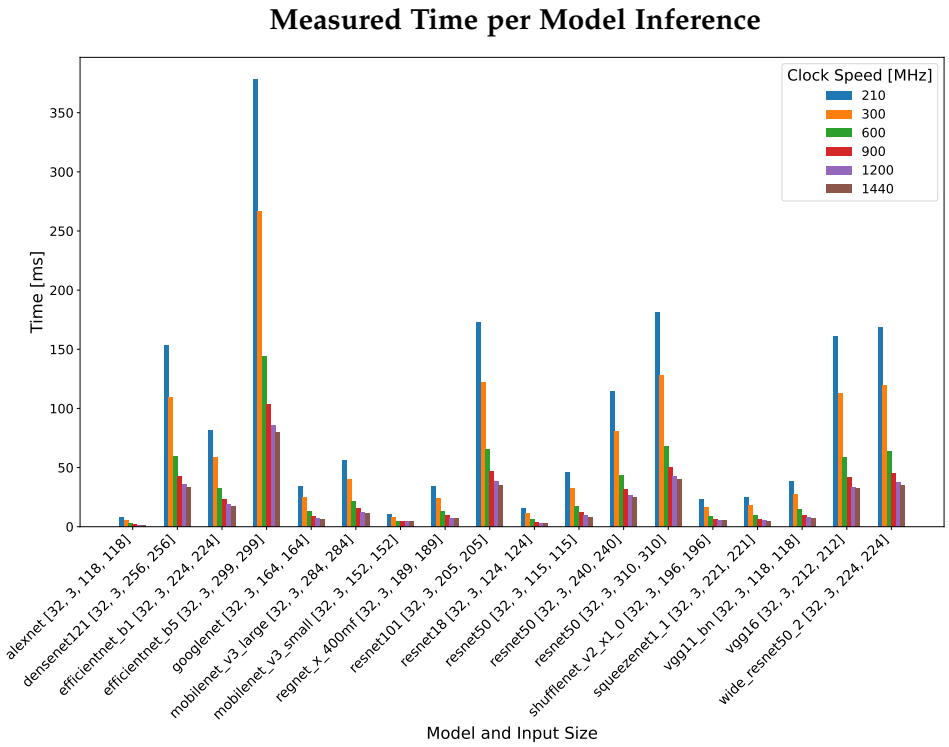


Figure 5.4: some descriptive caption

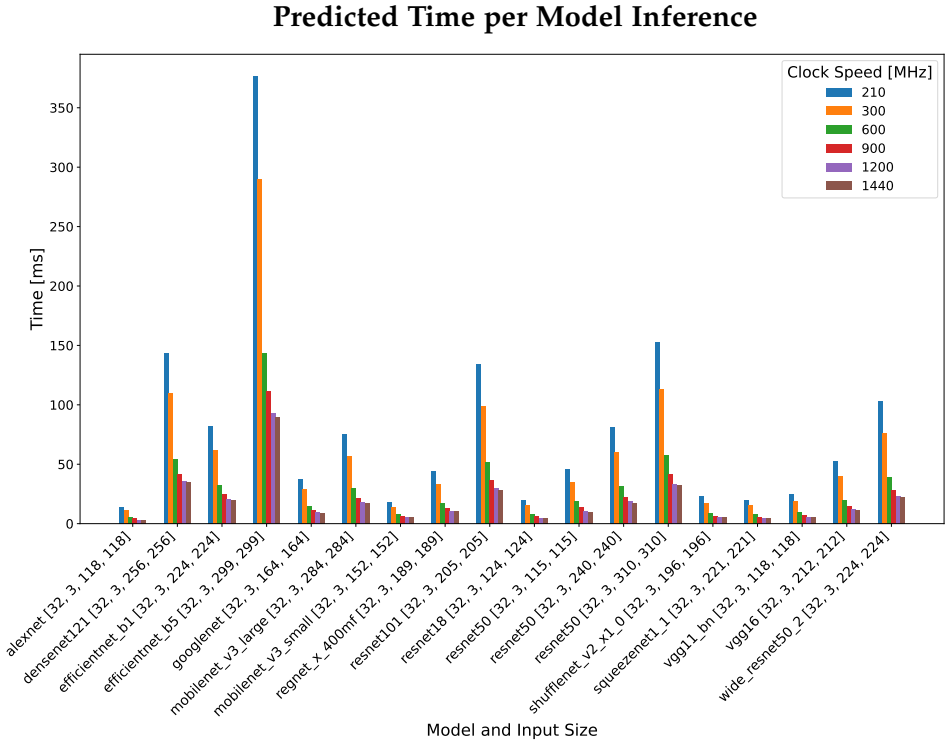


Figure 5.5: some caption

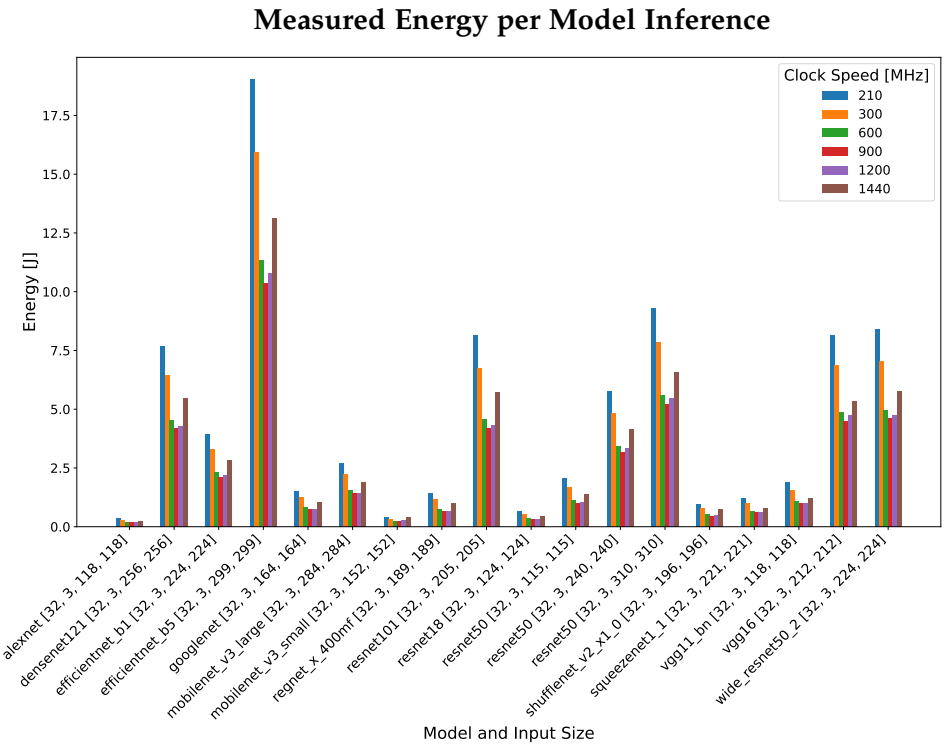


Figure 5.6: some descriptive caption

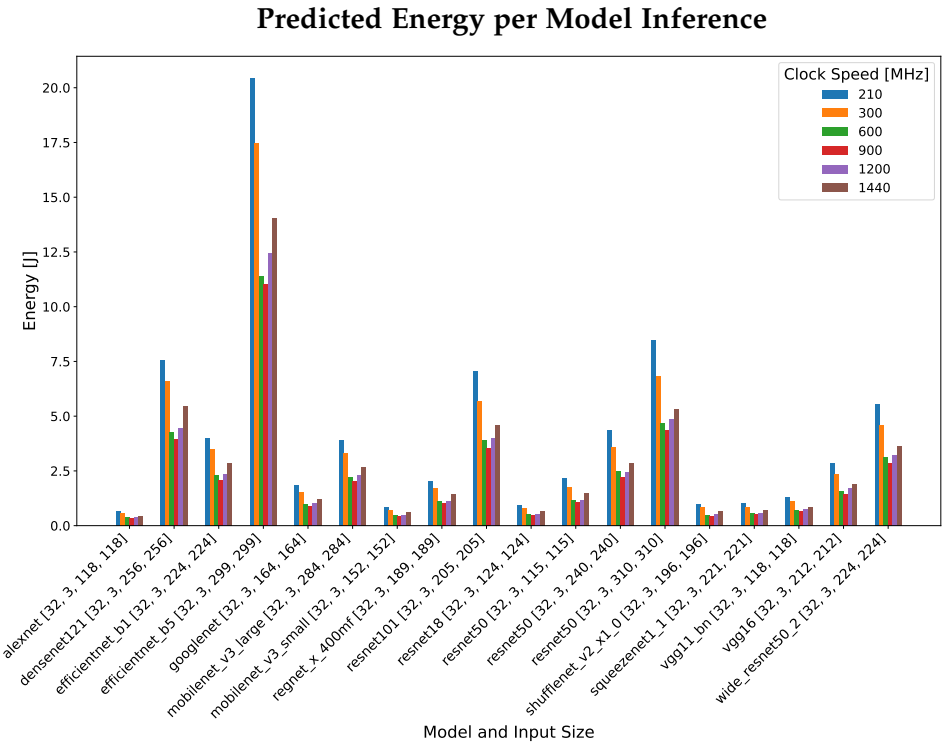


Figure 5.7: some caption

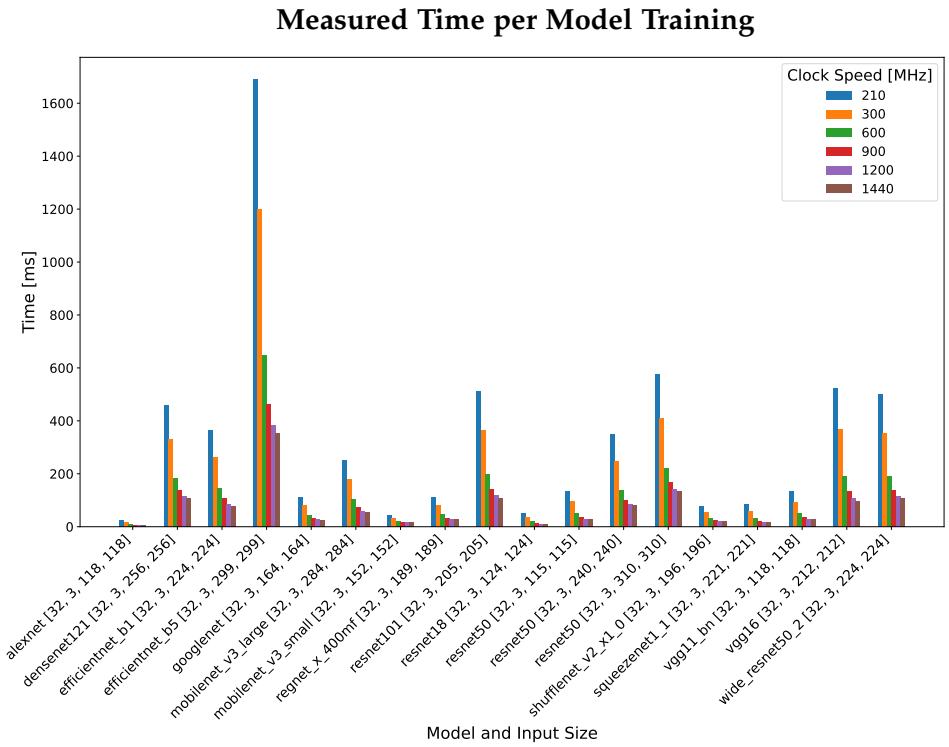


Figure 5.8: some descriptive caption

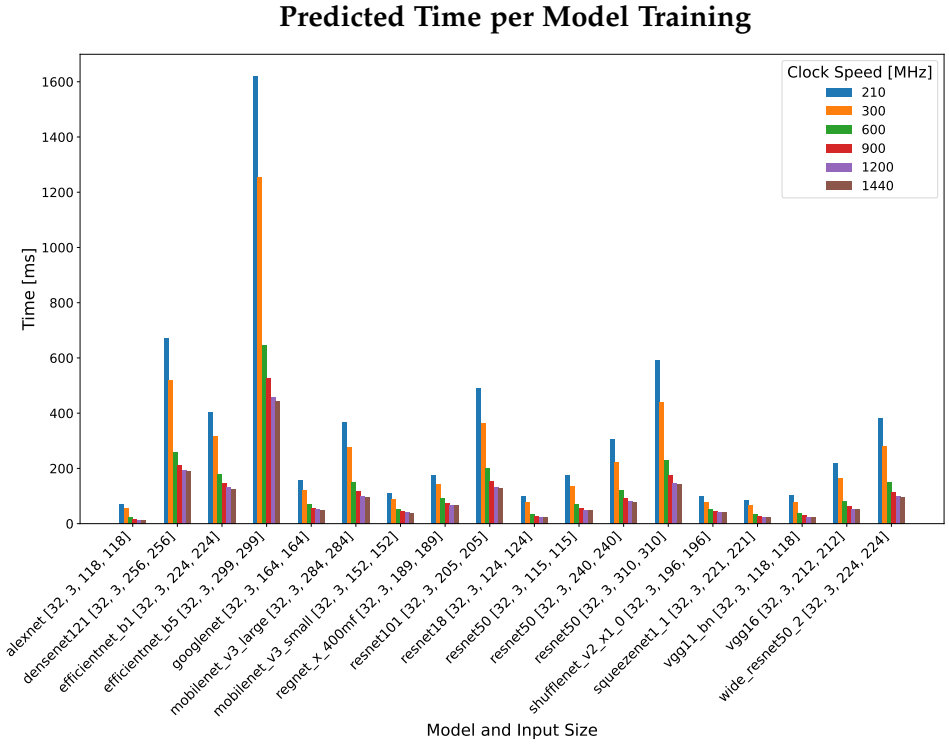


Figure 5.9: some caption

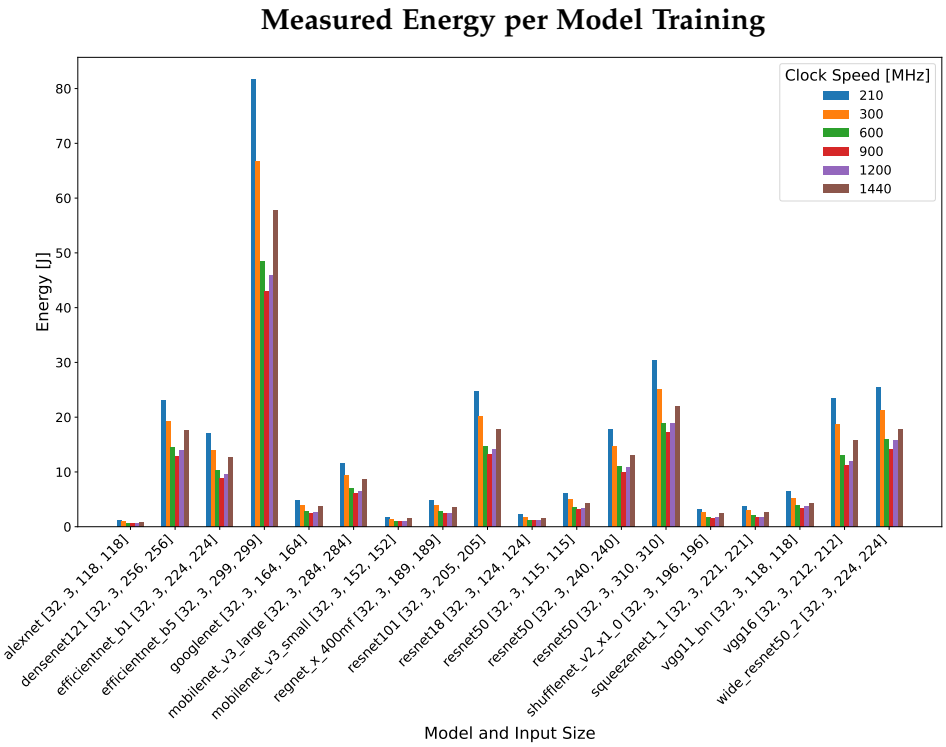


Figure 5.10: some descriptive caption

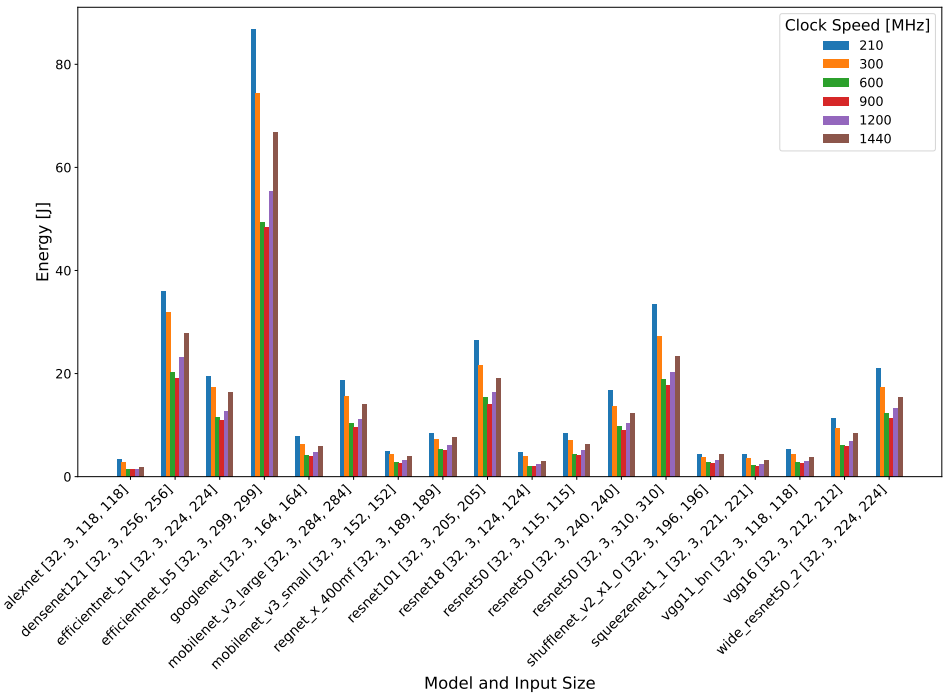


Figure 5.11: some caption

6 DISCUSSION AND OUTLOOK

Summary and discussion of the results and outlook/future work.

a | APPENDIX

Appendix here

ERKLÄRUNG

Ich versichere, dass ich diese Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den Day/Month/Year Here

_____

Constantin Nicolai