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

The fast broader adoption of ML applications has caused a surge in energy costs, necessitating a better understanding of tradeoffs between time and energy for ML performance. Previous work was focused on time-only or inference-only studies. We contribute: (1) time and energy profiling across inference and training, (2) performance prediction trained on our profiling results and (3) graphical evalutation and validation of profiling and predictor results. Profiling results of A30 performance across several core clocks reveal an energy optimum of 900 MHz, aligning with the manufacturer base clock of 930 MHz. This work provides the tools, which enable the correct choice of target GPU and clock speed for existing and future models.

ZUSAMMENFASSUNG

Die rasche Verbreitung von ML Anwendungen hat zu einem starken Anstieg der Energiekosten geführt und damit ein besseres Verständnis der Tradeoffs zwischen Zeit und Energie notwendig gemacht. Der Fokus vorheriger Arbeiten liegt auf reinen Zeit oder reinen Inference Studien. Unsere Contributions sind: (1) Zeit- und Energieprofiling für Inference und Training, (2) Vorhersagen trainiert auf den Profiling Ergebnissen und (3) eine grafische Auswertung und Validierung der Profiling und Verhersage Ergebnisse. Messresultate der A30 Performance über einige GPU Clocks zeigen ein Energieoptimum bei 900 MHz, welches sich mit dem Standard Base Clock von 930 MHz deckt. Diese Arbeit gibt uns die Werkzeuge um die richtige GPU und den richtigen clock speed für aktuelle und zuküftige Modelle zu wählen.

Nice quote here.

— Some Author

ACKNOWLEDGMENTS

Your acknowledgments here if desired

CONTENTS

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

The intended scope for this thesis is collecting a dataset, validating the usability of said dataset, using it to perform predictions and lastly evaluating the quality of the predictions.
In practice there are also limitations of the scope in terms of which hardware and which neural networks are included in the study. Here the hardware is limited to the Nvidia RTX 2080 TI and the Nvidia A30. The neural networks in use are all part of the Pytorch Torchvision library.
Lastly, due to the grid-search like nature of the study, there are points in the grid where some incompatabilities occurs. This may be a model which does not fit onto a GPU or a benchmark which takes a prohibitive amount of time for a specific setting. Given the amount of troubleshooting necessary to find workarounds for all these edge cases of little practical applicability, we take the freedom of not taking them into the scope of this work.

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

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

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

2 STATE OF THE ART AND RELATED WORK

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 other 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. Power readings are collected directly through nvidia-smi. While this limits our work to Nvidia GPUs, this 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 multiple 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 the most general objects which have 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 power 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 operation for

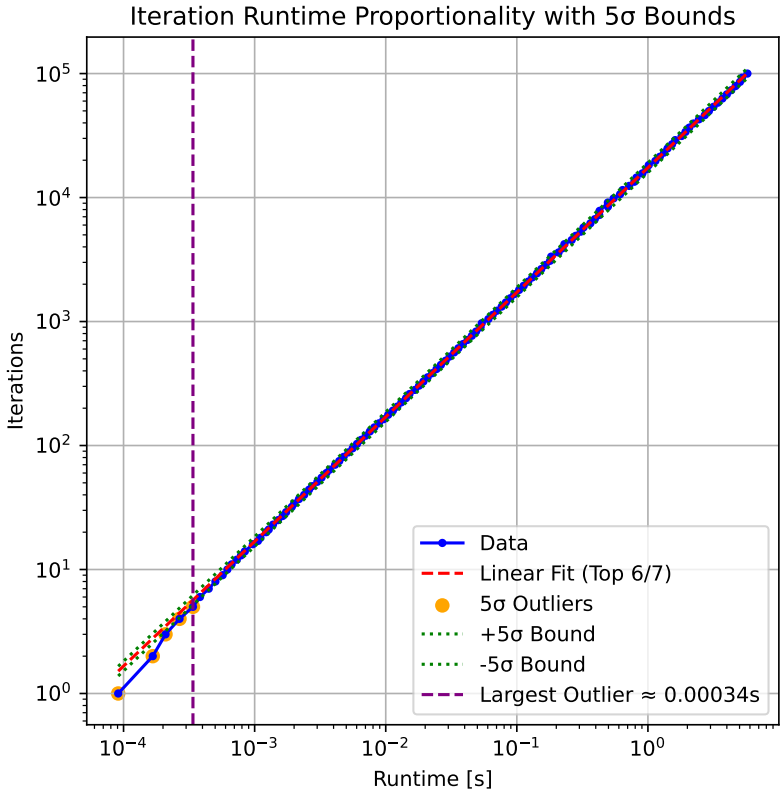

Figure 3.1: This plot shows the proportionality of the runtime with the number of iterations ran for an operation. A linear fit is performed to the $\frac{6}{7}$ of data points with most iterations. From this fit the standard deviation for the linear section is found. This allows us to identify the largest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

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 parameters.
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 are 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 measurement pipeline, some preparations and processing after the fact are necessary to ensure robust results.
As can be seen in Figure 3.2, 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 3.3 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

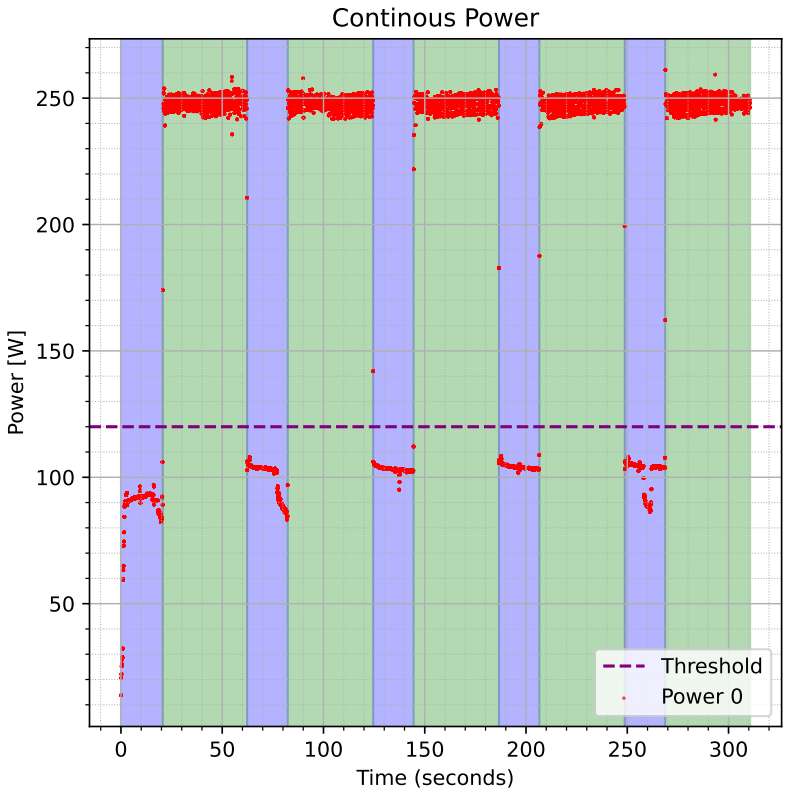

Figure 3.2: Power log on the RTX 2080 TI for alternating sleep and benchmark calls. Benchmark and sleep are marked as coloured sections. At each transition there are a few readings in between the two steady states. Those are the startup effect, that we are filtering out by the use of a 3σ channel around the initial mean and dropping all readings outside.

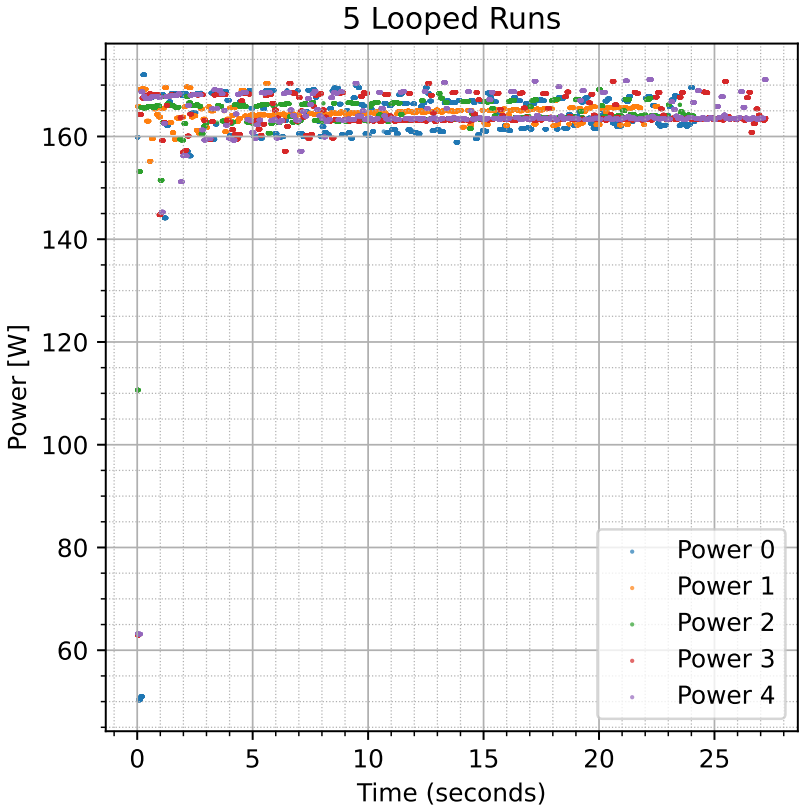

Figure 3.3: Power logs on the A30 for five runs of the same benchmark. They are overlayed to illustrate the reproducable pattern. Since the benchmark was run in the loop, we conclude the startup effect is not a result of the idle time introduced by the sleep calls in Figure 3.2, but is also present in our actual measurement script.

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 PROFILING 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 third 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`.
In the next step 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 the time and enery per iteration with their respective errors, as well as the average power.

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 additional 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 relationships between clock speed, runtime and energy consumption.

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 our energy prediction.
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.
A similar attempt was made with Extreme Gradient Boosting. This did result in comparable to slightly improved prediction performance compared to the random forest model. However, we expect increased training time for this method and only discovered the slightly improved performance very late in the process. For these reasons we did not switch the complete evaluation to a new model at that point. We would recommend further investigation into using this method for future work.

4.2 PREDICTION ARCHITECTURE

As we are using the `sci-py` implementation of the random forest model, we did not have to create the 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 models only use numerical inputs, we made sure to 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 arose 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. IWe do however want to support all operators in one predictor model, which in turn means, we needed 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 applicaple 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. For the flags, a 1 signifies the entry being applicaple, while a 0 signifies it not being applicable.
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 a 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 attributes accordingly. With the heuristic approach used here, we have attempted to err on the side of rather including more attributes then fewer ones.
The last important input vector entry we need to mention is the GPU clock speed. Even though the initial implementation trained runtime and power 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 negative impact on our prediction

accuracy. This way it is just another parameter 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 either the inference or the training predictor.

4.3 PREDICTION WORKFLOW

As apparent from the architecture, our prediction model operates on the operations level. In case this is what is desired for a given study, the current state is already sufficient. 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 full neural networks. In that case the workflow is very similar to the dataset creation. It starts with extracting the operations and how often each operation occurs. Afterwards all unique operations will be run through the predictor and the results will be summed up according to their number of occurences within the model.
Critically, due to the modular nature of the approach, there is no need for the neural network being predicted to be actually executed or even exist in functioning form. As long as the operations are available 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 present its results. By providing quantitative results we can validate the methodology and provide an informed impression of both its capabilities and limitations.

5.1 DATASET VALIDATION

In this first section our focus is to ensure the datasets we collect for training are reasonable accurate. This way we prevent basing our predictions on a training set which is skewed from the very start.

5.1.1 Methodology

Our approach to validating the datasets is conceptually rather simple. Our measurements of the individual operations should resemble their execution in a DNN. In order to verify this, we compare our measurements of complete DNN executions, to the result summed up from our individually measured operations.
Our measurements for the complete DNN executions are performed using the same pipeline used to measure the individual operations. Using this pipeline is possible because the script can simply view a complete DNN as one larger operation to be profiled.
We also built the script that extracts which unique operations are present in a speficic DNN in a way that is also tracks how often each unique operation occurs. This way, we can sum the results from our collected dataset accordingly.

5.1.2 Hardware Platforms

We begin by looking into our findings for our three original GPU configurations. In a later part we will look into results for 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.
Given the capability of floating point 32 computation on the A30's tensor cores, we decided to probe its performance characteristics between having its tensor cores enabled and disabled. We cannot make the same differentiation for the 2080TI, because its tensor cores do not support `FP32`, which we use in all our benchmarks.
This leaves us with three configurations for the dataset validation. The 2080TI with default settings, the A30 with default settings and the A30 with its tensor cores disabled.

5.1.3 Results

5.1.3.1 RTX 2080 TI

Our results for the 2080TI our findings are not perfect. Agreement between measured and summed results does look rather promising for larger model-input sets. However, for smaller ones, there are instances where the agreement 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 overestimates both runtime and energy. However, the overestimation is more pronounced for runtime than for energy.

5.1.3.2 A30 Tensor Cores Disabled

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

5.1.3.3 A30 with Tensor Cores Enabled

Our results for the A30 with its tensor cores enabled are very promising. While the earlier trends did not completely vanish, they are even less pronounced than for the A30 with disabled tensor cores. This hardware config yielded the most precise summation of the three configurations.

5.1.3.4 Runtime Uncertainty Behaviour

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.

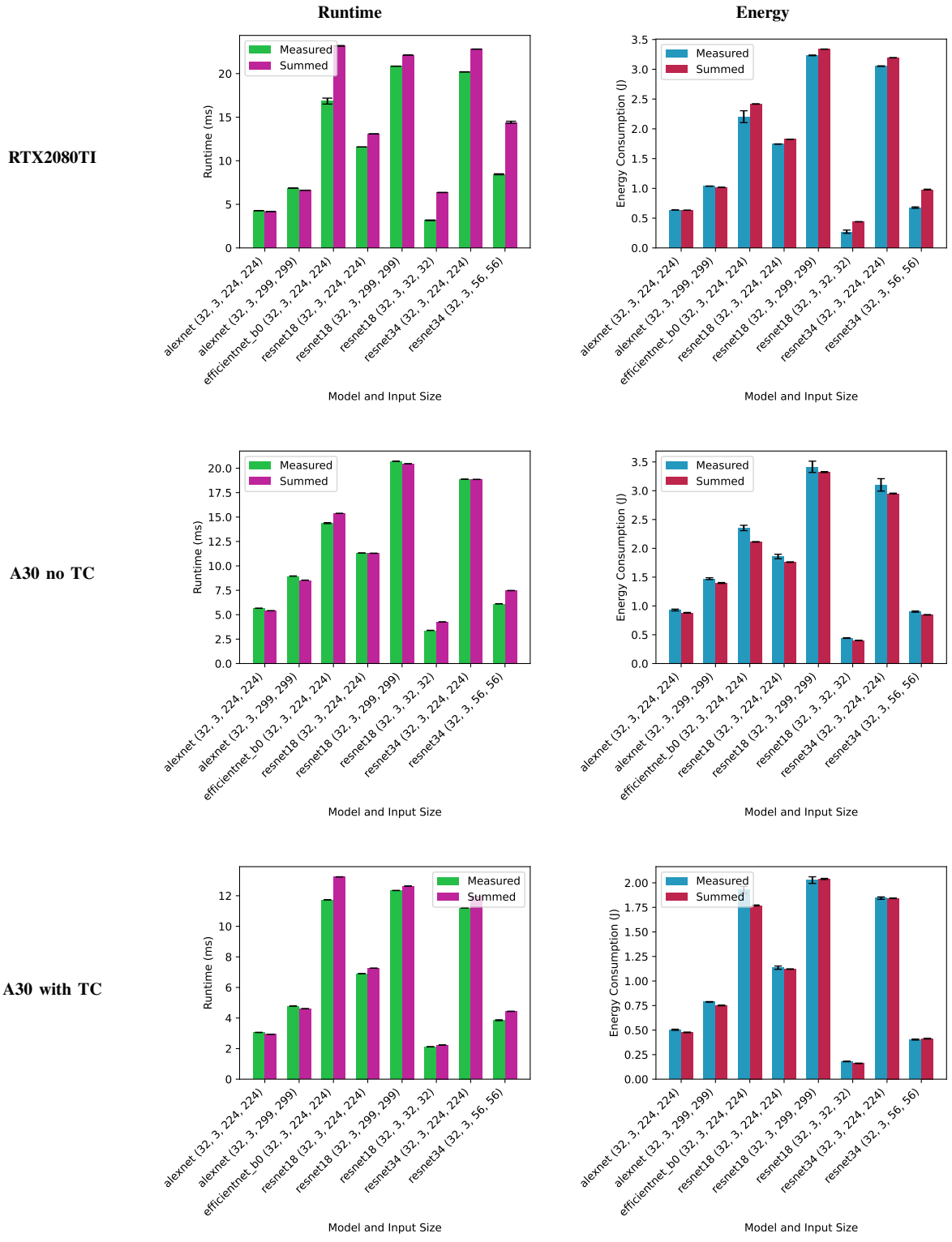

TABLE I: These are the plots for smaller model-input sets. The ones in the left column show the runtimes we measured with full model-input runs compared to our approximations. The ones in the right column show the same comparison for the energy consumption. The approximations were obtained by summing 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.

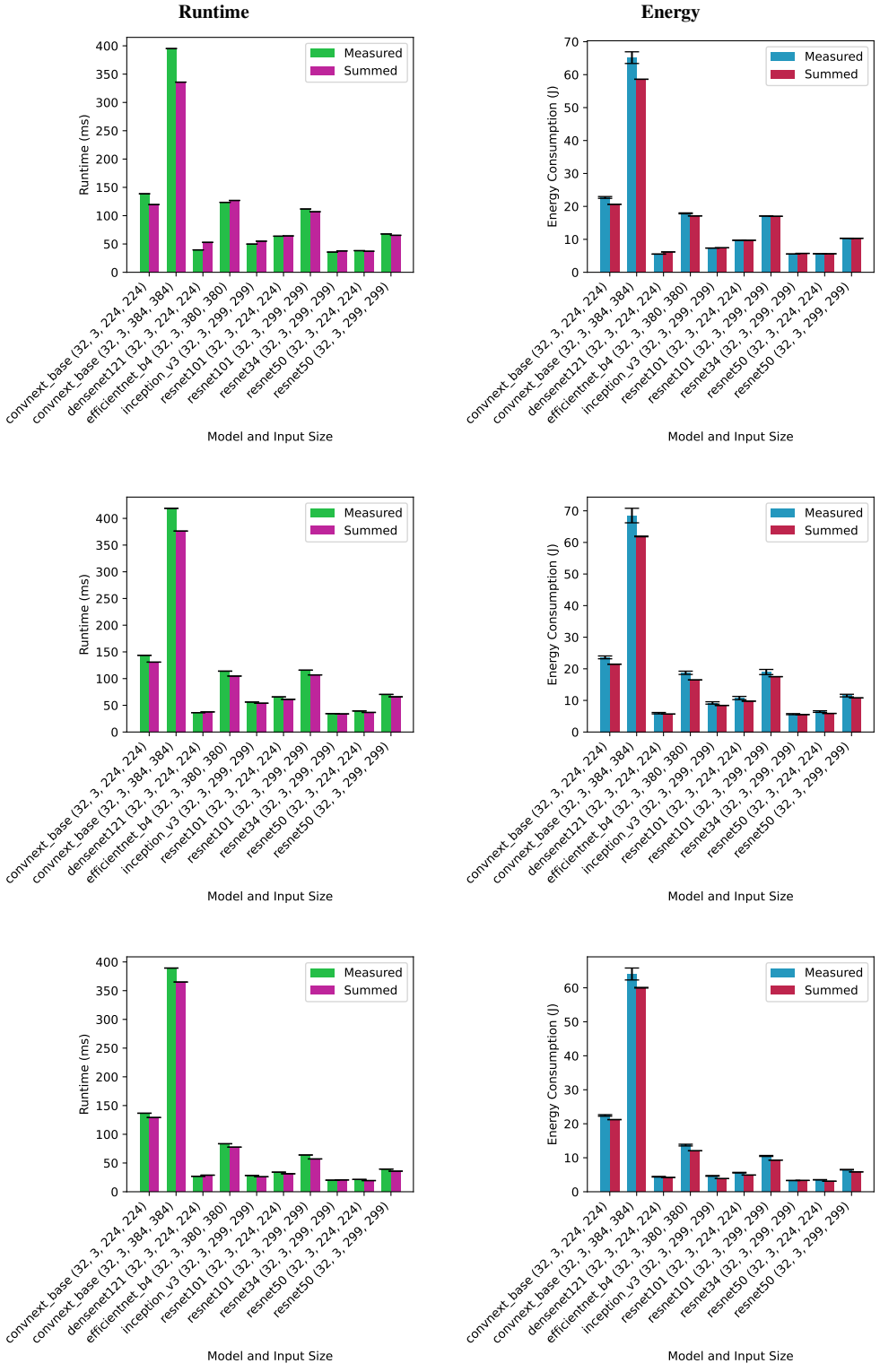

TABLE II: These are the plots for larger model-input sets. The ones in the left column show the runtimes we measured with full model-input runs compared to our approximations. The ones in the right column show the same comparison for the energy consumption. The approximations were obtained by summing 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.

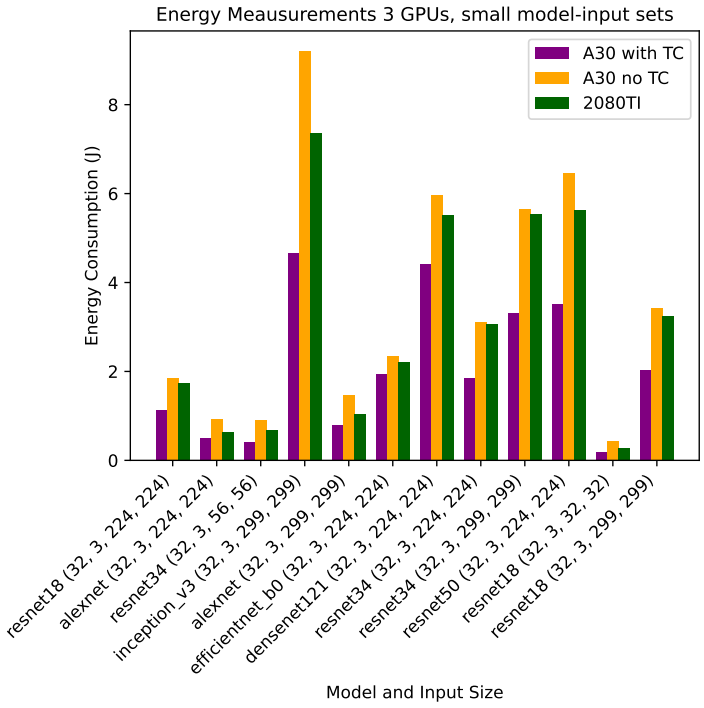

Figure 5.1: Comparison of energy measurements for the 2080TI and the A30 with tensor cores once disabled and once enabled. The resulting ordering is identical for all model-input sets. However, the relative differences show a lot of variation, being more pronounced for these smaller model-input sets.

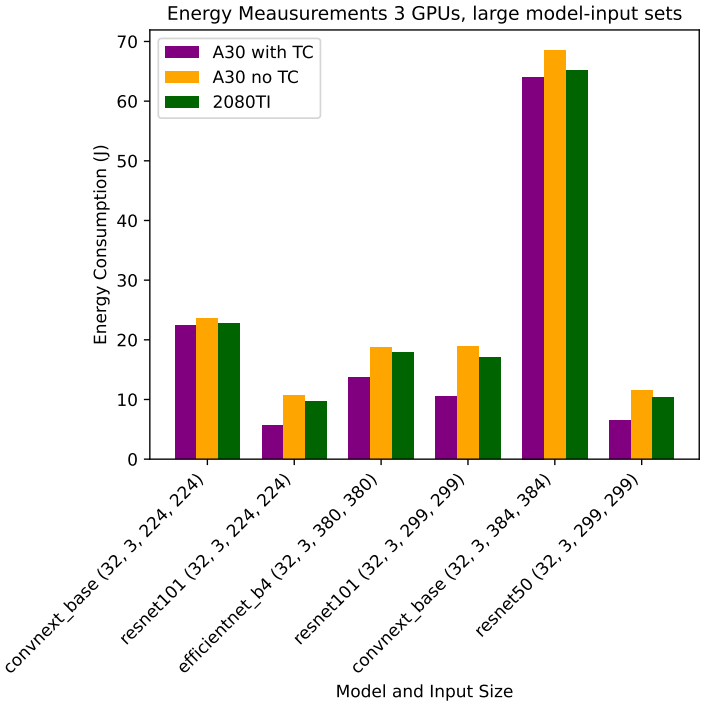

Figure 5.2: For these larger model-input sets, we maintain the ordering, but we observe far weaker relative differences between the hardware configurations. For model-input sets with a high dependency complexity, like the ConvNext Base model, we find the most similar energy results across the configurations.

5.1.4 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-

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. In order to provide a more well rounded representation of the results, both the R^2 score for the previousy unused test set, as well as the mean R^2 score for a 15 fold cross validation are given.
Because the inference and training datasets are conceptually seperate and measured independently, the prediction models are seperate as well. Naturally, their evaluation is split up too.

5.2.1.1 *Training A30*

Training A30	**Random Forest**	**XGBoost**
CV $\overline{R^2}_{time}$	0.876 ± 0.073	0.912 ± 0.04
Test Set R^2_{time}	0.9036	0.9018
CV $\overline{R^2}_{power}$	0.977 ± 0.007	0.985 ± 0.003
Test Set R^2_{power}	0.9797	0.9868

Table 5.1: The operations level results for this predictor are very good. With scores of around 0.9 for the runtime predictor and around 0.98 for the power predictor, its performance is very respectable. The XGBoost predictor shows marginally better preformanceto the random forest one, but the results are vert close.

The resulting values for the model predicting training performance can be found in Table 5.1. Keep in mind that the results do show a small amount of variation depending on the random seed used for the predictions models and the test set, training set split.
As can be seen from the results in the table, the XGBoost model preforms marginally better in terms of prediction accuracy. Unfortunately, this was only discovered very late in the research process, because it was hidden in the random seed uncertainty. As further work as already done with the random forest model, a pivot back would go beyond the scope. Additionally, the prediction performance may be worse, but not by a large margin. Therefore further results shown were acquired using the random forest model. This also provides the advantage of using the model which is more simple in nature and should exhibit better scalability properties in general. Investigations into which model can provide predictions with lower latency could also be part of future work.
We also included Figure 5.3 in order to convey a visual impression of the prediction performance. For a random subset of operations from the test set, it shows the predicted value alongside the measurement value.

Both the R^2 score results as well as the visual interpretation tell a similar story. We clearly have a stonger predictive power for the power model than for the runtime model. The runtime model still provides decent results, but they are not on the same level as the power model. Both prediction performance and prediction latency requirements will determine whether this approach is suitable for any given application in the end, but it seems reasonable to assume that the current performance would be applicable to some areas.

5.2.1.2 Inference A30

The resulting values for the model predicting inference performance can be found in Table 5.2. Our resulting R^2 scores lie in a very similar regime as the ones for the training model.
If we were to attribute the lower predictive performance of the runtime model with XGBoost and for the random forest model test set score to more than the inherent uncertainty, it might be explainable by the smaller actual runtimes in the inference case. Smaller runtimes with the same measurement methodology should result in similar absolute uncertainties and therefore in larger relative uncertainties. These larger relative uncertainties might play a part in the slightly weaker predictive performance we observe.
All power model scores both for the random forest model, as well we the XGBoost model show practically identical performance to the training model, taking into consideration the additional uncertainy introduced by the random seed affecting the predictive performance slightly.
Given the very similar results, both in these metrics and in the visual interpretation of Figure 5.4, our conclusions for the inference model are the same as for the training model. The lower predictive power of the runtime model compared to the power model will result in a limit of its suitable applications at some point, but its performance is still acceptable.

Inference A30	**Random Forest**	**XGBoost**
CV $\overline{R^2}_{time}$	0.890 ± 0.66	0.898 ± 0.05
Test Set R^2_{time}	0.8827	0.8924
CV $\overline{R^2}_{power}$	0.981 ± 0.003	0.988 ± 0.002
Test Set R^2_{power}	0.9853	0.9895

Table 5.2: Our inference predictor for the A30 does very well. With runtime sccores just below 0.9 and power scores higher than 0.98, it is performs just as well as the A30 training predictor.

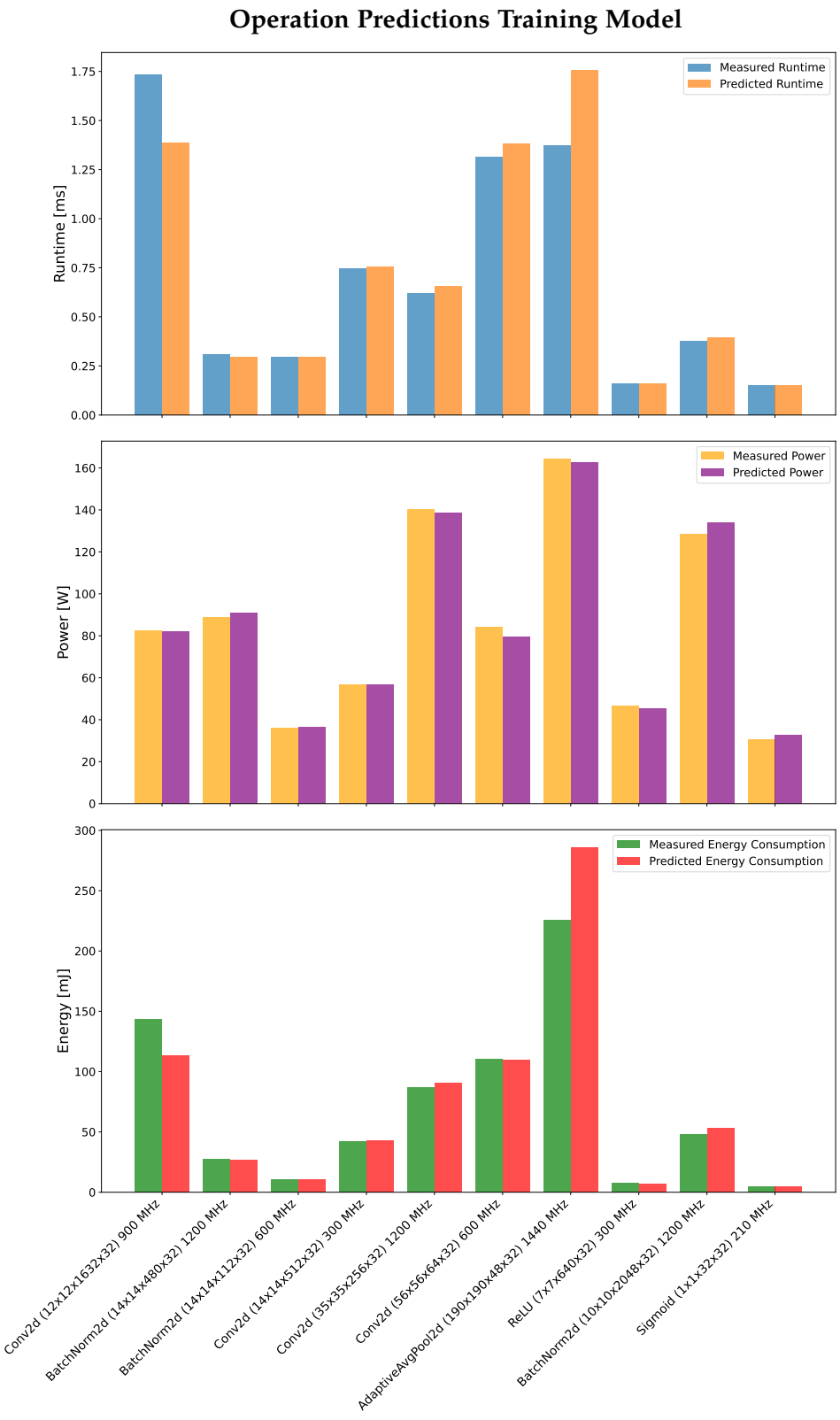

Figure 5.3: Comparison of predictions to measurements for ten operations from the test set for the training prediction model. While we have outstanding agreement for most of them, we can see the limits of our predictive power in the results for operations 2 and 9. This illustration can give us a more intuitive impression of the R^2-error of 0.87 for the runtime and of 0.97 for the wattage.

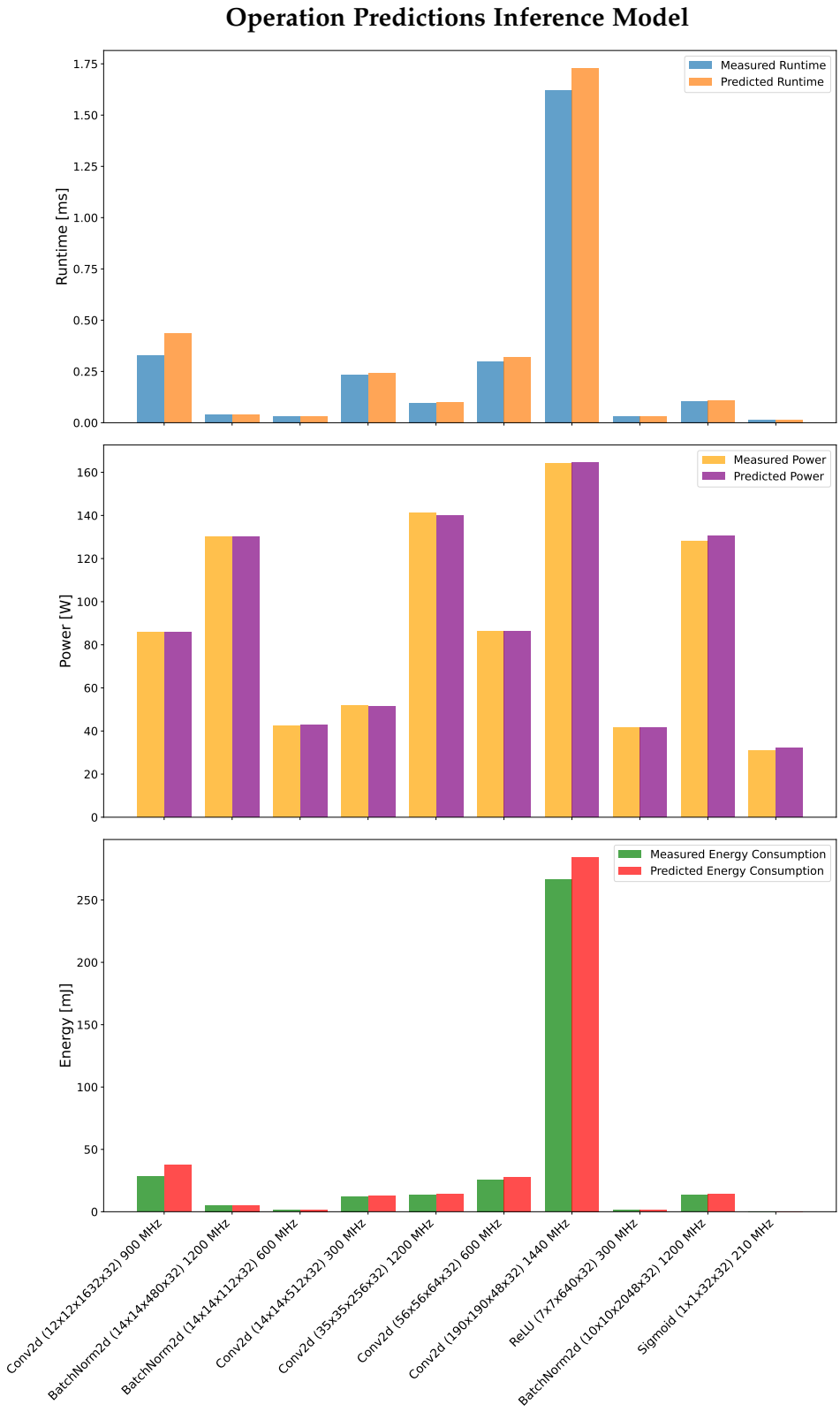

Figure 5.4: Please write me.

5.2.1.3 Training RTX2080TI

The prediction model for the RTX2080TI is not built to predict for specific clock speeds. Instead it is trained on a dataset collected running the GPU in its default configuration. Therefore that is also what it predicts for. Unfortunately, we find worse prediction performance than for our A30 predictors. The results can be seen in table 5.3. They are not unusably bad, but disappointing in comparison. Interestingly however, this predictor breaks the pattern of XGBoost always performing a little better. Here we find a few cases where is performs better and a few where it performs worse. This might come down to the property of random forest models of generalizing pretty well, even with a smaller amount of training data. Since this is a dataset for the default auto clock speed, it is several times smaller then the dataset for multiple clock speeds.

Training 2080TI	Random Forest	XGBoost
CV $\overline{R^2}_{time}$	0.755 ± 0.122	0.684 ± 0.359
Test Set R^2_{time}	0.813	0.793
CV $\overline{R^2}_{power}$	0.920 ± 0.058	0.921 ± 0.054
Test Set R^2_{power}	0.866	0.869

Table 5.3: The training predictor for the 2080TI scores notably lower than the A30 predictors. It also reverses the A30 predictor trend of the XGBoost models performing slightly better. For the 2080TI predictors, XGBoost performs a little worse. The smaller training set due to the lack of a clock speed study for the A30 and the smaller focus on consistency of a consumer GPU compared to the A30 might contribute to the weaker predictor performance here. Still, with a runtime score of over 0.75 and a power score of around 0.9 it is far from unusabele.

5.2.1.4 Inference RTX2080TI

The resulting R^2 scores for our 2080TI inference predictor can be seen in table 5.4. Especially for between the runtime models we see a steep drop in R^2 score from the random forest model to the XGBoost model. But even for the better performing random forest model, this is still the worst performing predictor between our four predictor models. This could caused by a combination of multiple contributers. It is trained on a smaller dataset, since it is not a multi clock model and it is an inference predictor, which means it has to predict smaller absolute values, which will have larger relative errors in the training set. Lastly, since the 2080TI is a consumer GPU and not a datacenter GPU, its behavior is tuned to focus on the best burst load performance, but not necessarily on the greatest consistency and stability compared to a datacenter GPU such as the A30.

Inference 2080TI	Random Forest	XGBoost
CV $\overline{R^2}_{time}$	0.735 ± 0.088	0.607 ± 0.079
Test Set R^2_{time}	0.769	0.613
CV $\overline{R^2}_{power}$	0.939 ± 0.034	0.944 ± 0.026
Test Set R^2_{power}	0.908	0.885

Table 5.4: The inference predictor for the 2080TI is our worst performer. It shares the challenges of the 2080TI training predictor, while having to work with a dataset of smaller values, which naturally have larger relative errors. Here the XGBoost predictor for runtime scores so much lower that there is not question in using the random forest one. Sticking to the random forest predictors here, we have a runtime score of 0.73 and a power score of 0.94, which is lower than we would like but still decent enough.

5.2.2 Neural Network Level

In this next step, we will evaluate the prediction performance of the random forest model on the neural network level. We will perform an more detailed ananlysis on this part, as this is the abstaction level which is most commonly used and therefore the most relevant to provide predictions for.
Since our preditction model is capable of providing predictions for different clock speeds, we will conduct the validation for each clock speed. As this is a graphical validation, we will provide the measured results for each clock speed and below the predicted results. This way both the direct comparison between measurement and prediction is visible, as well as the comparison of the behavior across different clocks compared between measurement and prediction.
Results are given for both runtime and energy, as well as for both inference and training. Additionally, a plot of the product of runtime and energy is provided in order to give an option which may show a tradeoff between optimizing for either on their own. For this the results for each clock speed are normalized, because the orders of magnitude between the different models became to large to remain readable in this product.
Furthermore, not as a preplanned point of interest for this work, but rather as a fortunate byproduct of the necessary measurements we have also found some patterns in the behavior accross clocks, which will will discuss for a bit.

5.2.2.1 Measurement Observations

While we are not surprised by the runtime measurement results simply exhibiting shorter runtimes for higher clock speeds, we do find an interesting pattern for the energy measurement results. Both for training and inference we observe a "U" pattern across the clock speeds, see

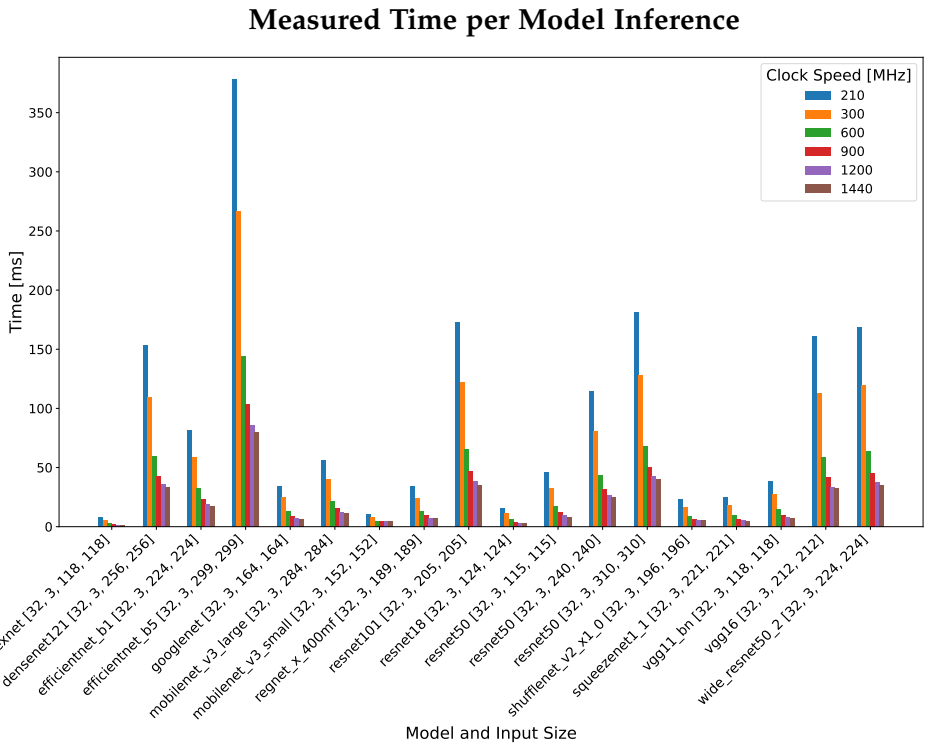

Figure 5.5: As expected, the runtime decreases with an increase in clock speed for all tested models.

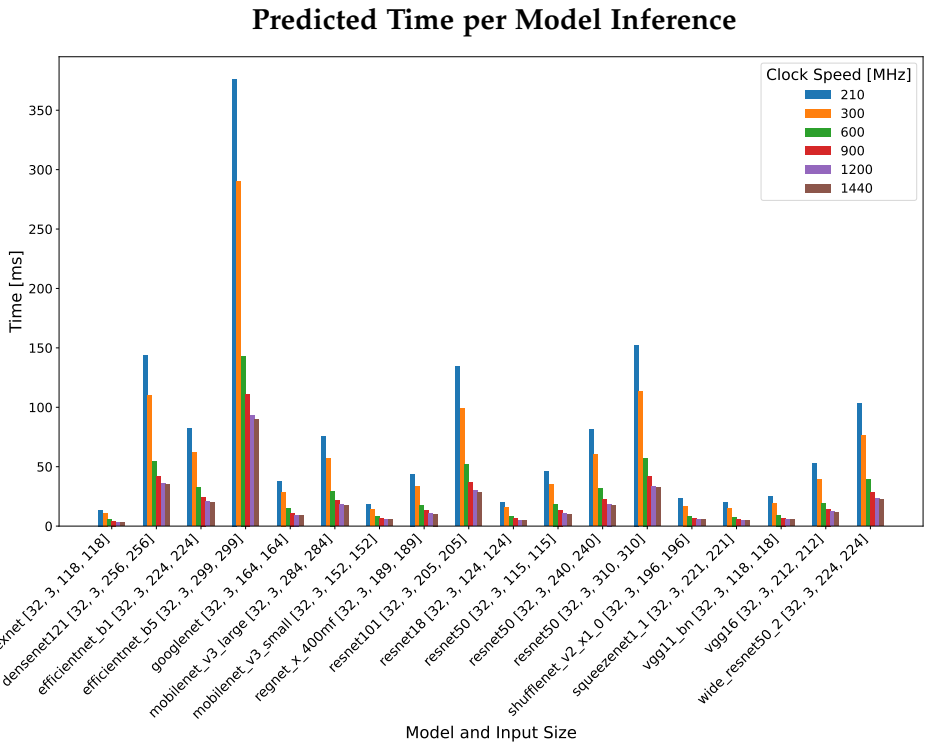

Figure 5.6: While the absolute results for the predictions may be close, but not perfect, the ordering between different clock speeds is maintained from the measurement results. This allows the predictions to be used to determine the optimal clock speed.

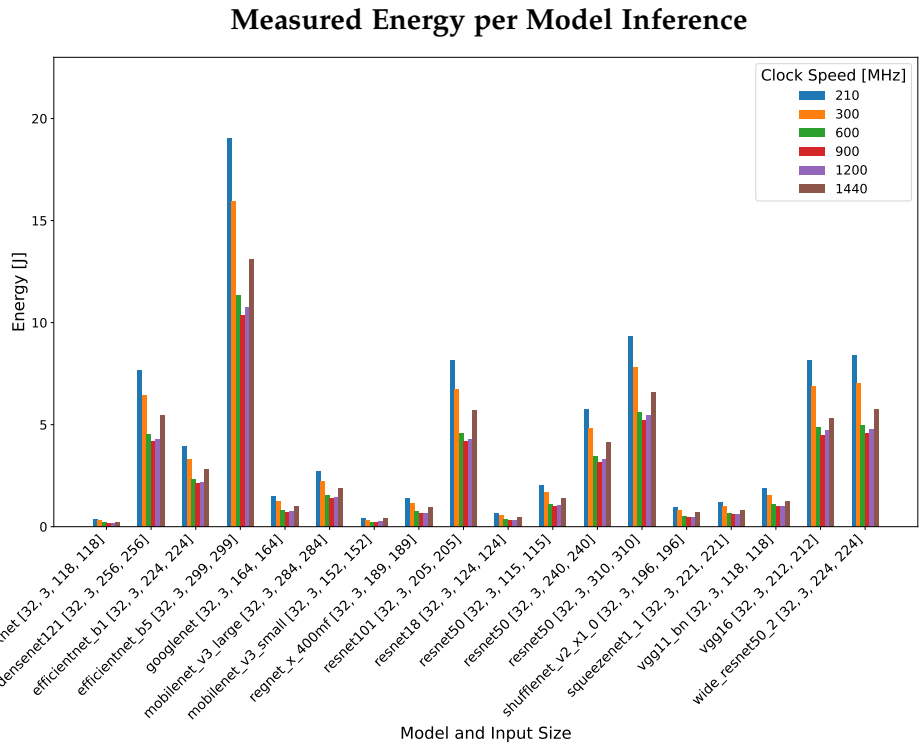

Figure 5.7: As opposed to the monotonous relationship between runtime and clock speed, for energy the optimal clock speed lies at 900 MHz.

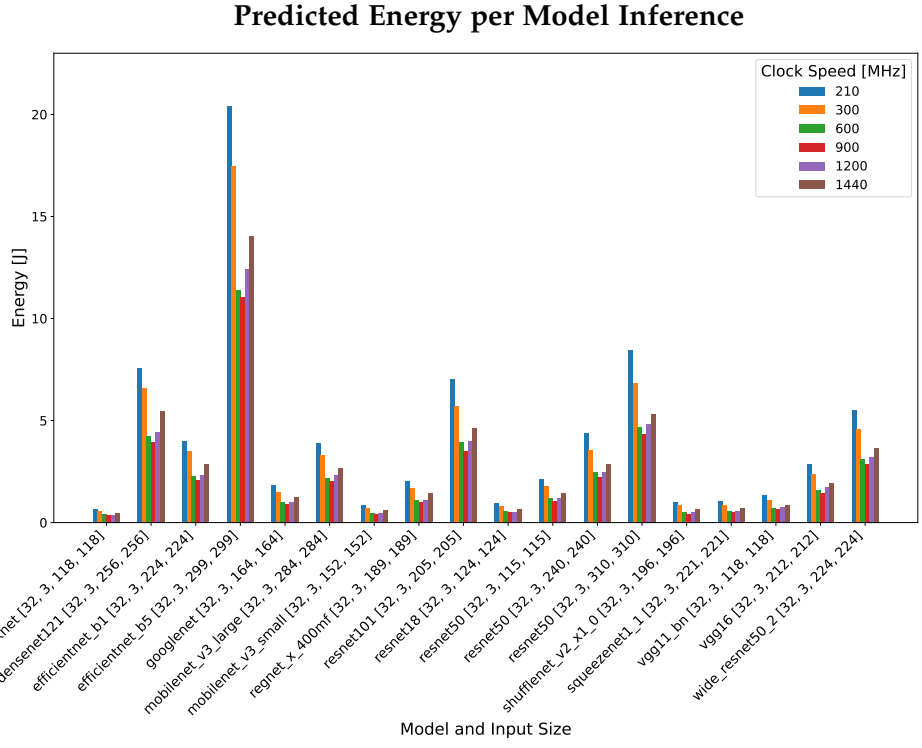

Figure 5.8: As expected from the operations level results, we can see that the predictions for the energy are even closer then the runtime results. Of course they also maintain the clock speed optimum found in the meausurements, making the predictions useful for energy optimizations as well.

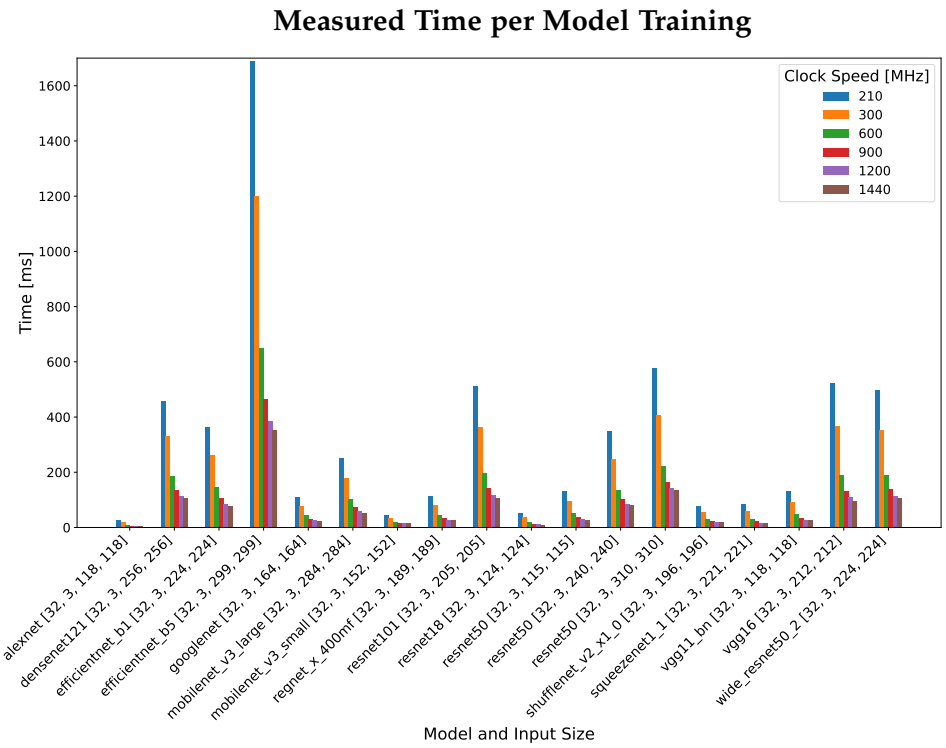

Figure 5.9: For these training runtime results we see the same behaviour as for inference. Lower clock speeds leed to longer runtimes.

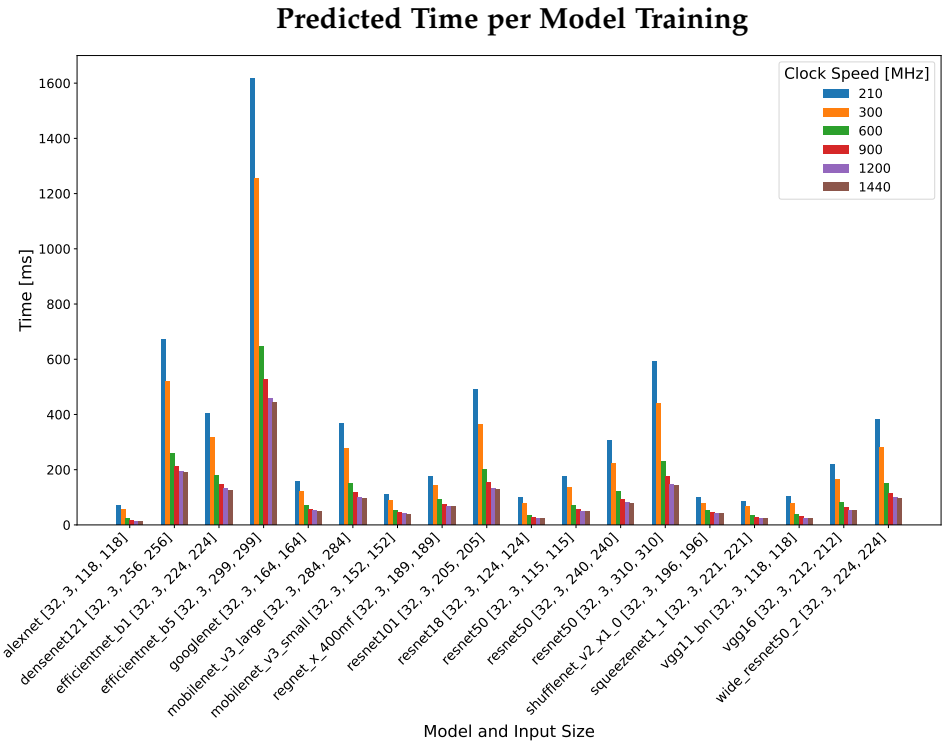

Figure 5.10: We see a similar prediction preformance to the inference case. The prediction provides the correct ordering, even though the absolute values are not perfect.

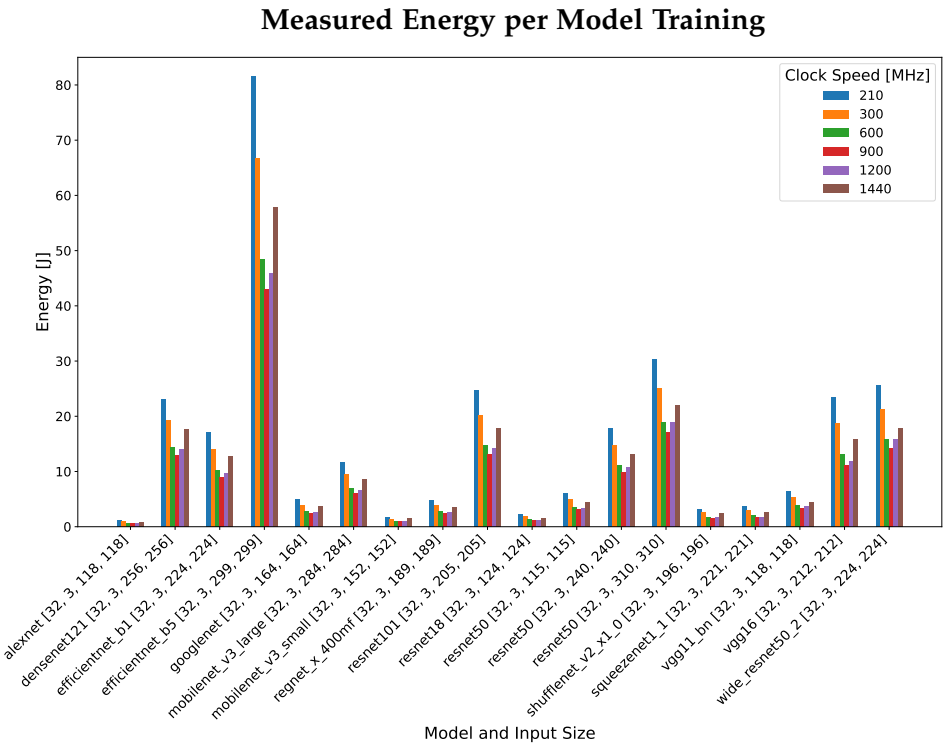

Figure 5.11: some descriptive caption

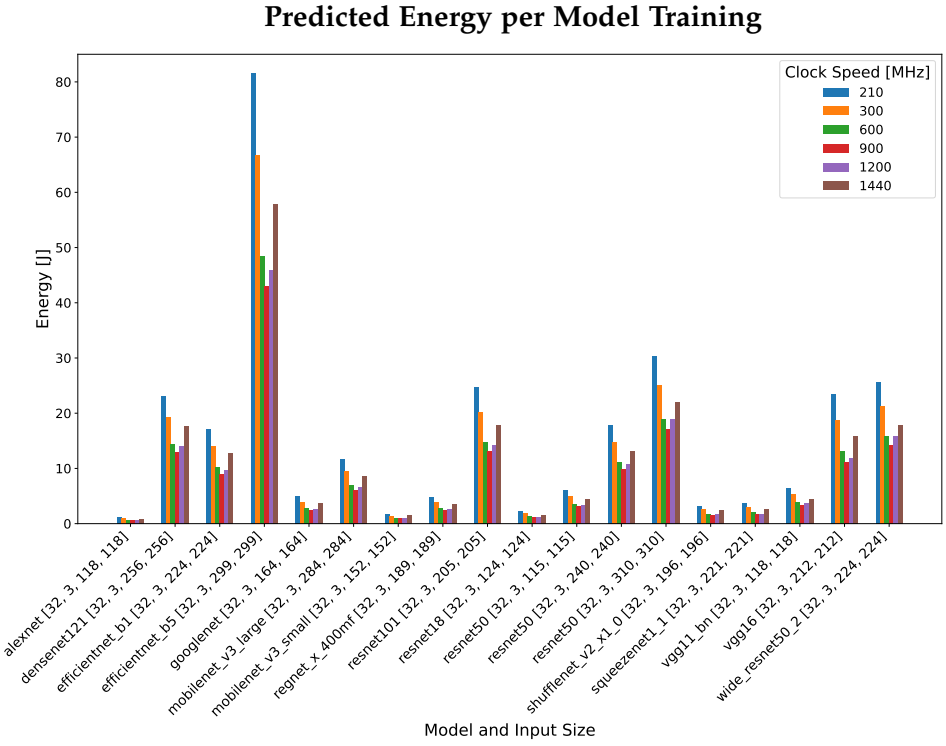

Figure 5.12: some caption

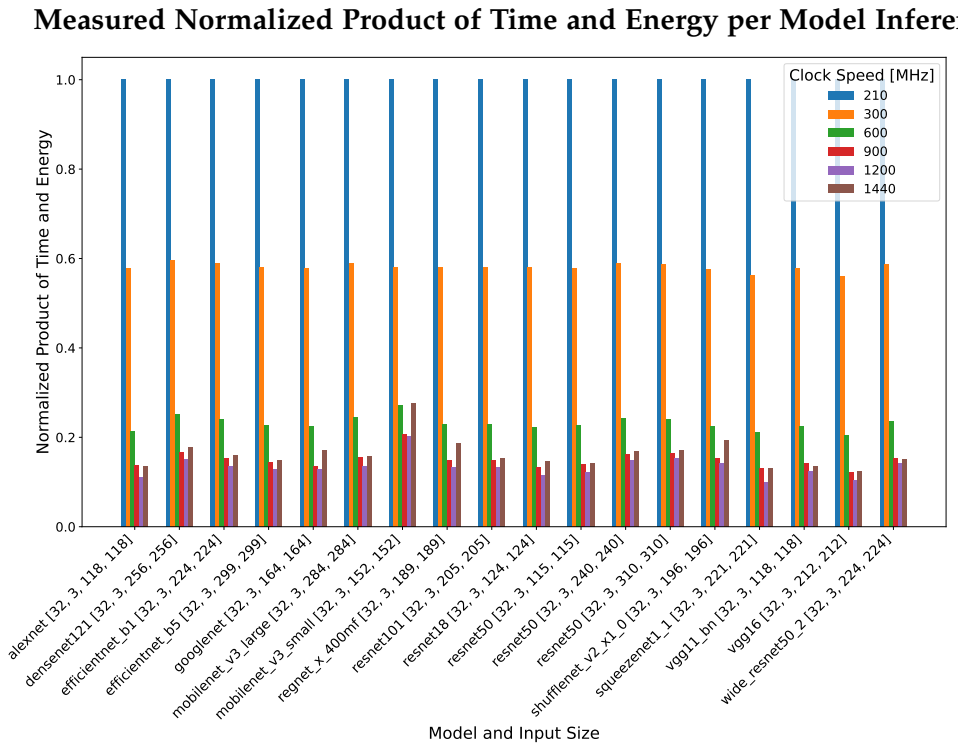

Figure 5.13: some descriptive caption

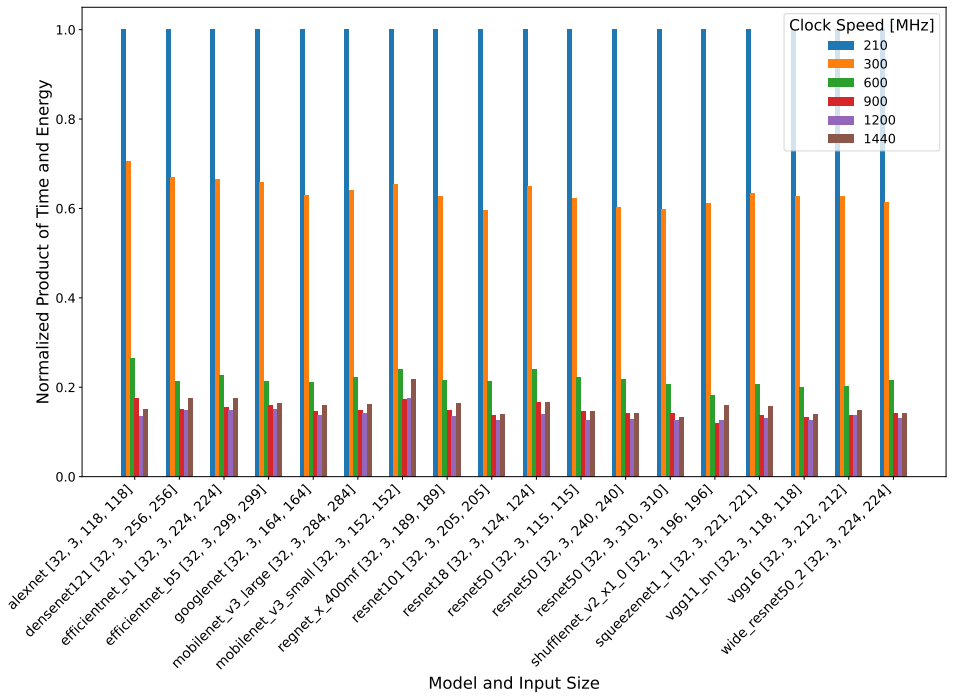

Figure 5.14: some caption

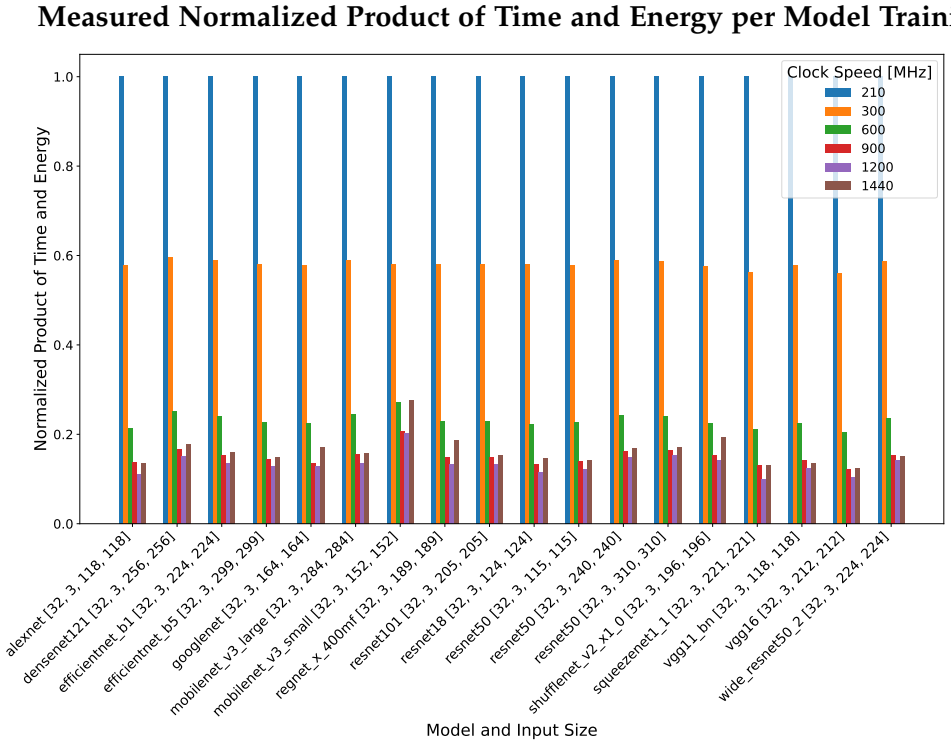

Figure 5.15: some descriptive caption

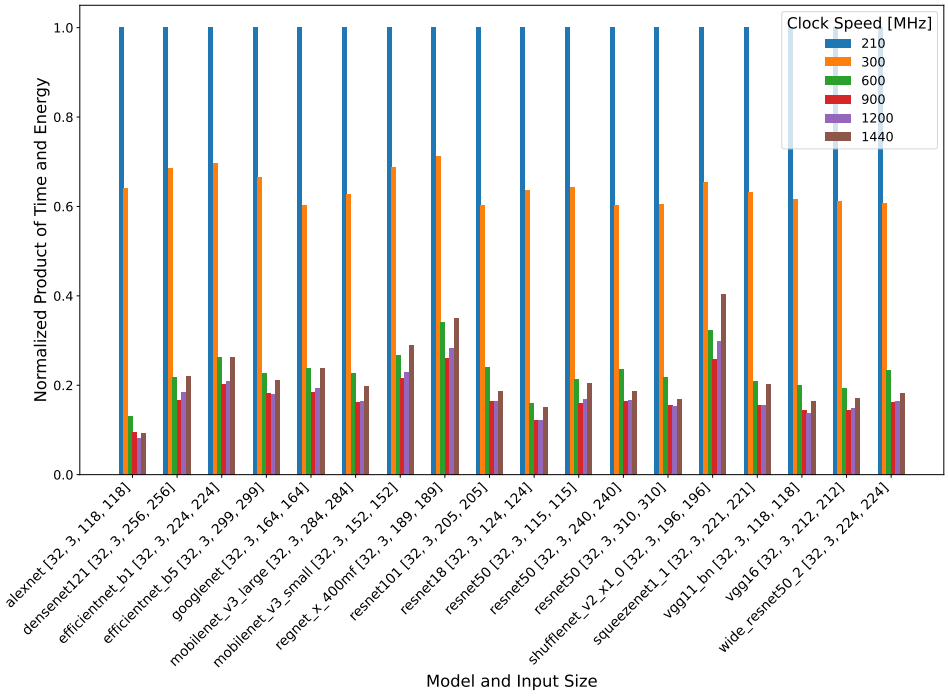

Figure 5.16: some caption

Figure 5.17. With very high energy costs for very low clock speeds and high energy costs for very high clock speeds as well. But in between, for higher, but not very high clock speeds, we see a minimum for the required energy at a clock speed of 900 MHz on the A30. More interestingly this pattern remains stable across both training and inference and for all models tested in this work. Therefore, even if there might be cases where it is not optimal, it is surely a good rule of thumb to use this as a starting point, when energy efficiency is of interest.
When trying to optimize for a balance between energy and runtime, the story becomes less clear cut. Looking at the product of runtime and energy, see Figure 5.13 and Figure 5.13, it becomes clear, that the optimum is not always at the same clock speed. However, within the clock speeds we tested, starting at 1200 MHz seems like a good bet, since it is the most common optimum for the product of runtime and energy.

5.3 PREDICTOR LATENCY

After some more tweaks to make the performance less abysmal, we arrived at a loading time of around 30 seconds for the full set of models used to validate te predictor. These 30 seconds are of in a completely different order of magnitude in terms of execution time compared to preproccessing, prediction of the models or even execution of the models. Realistically, this time is more comparable to having to download a model before being able to benchmark it. The next largest contributer in terms of execution time is preproccessing the dataset in order to format is in a way that can be used as an input for our predictor model. In its current form this step takes somewhere around 500 ms. For the full validation set of 18 model-input sets it takes around 600 ms. For an individual model it takes less than 500 ms. The fact that this step is relatively speaking so slow if unfortunate. This is a result of this being research code which is not optimized. While it is not possible to predict how much headroom for optimization there is, some improvements should be possible, if this were to go to production.
The most important evaluation in terms of predictor execution time is however the execution time of the actual predictor model. Predicting time and energy at a specific clock speed for all model-input sets in the validation set shows very good performance. Across multiple clock speeds and both for the training predictor and the inference predictor the resulting execution time hovers around 30 ms. Even for the scenario with the smallest execution time, inference at maximum clock speed, this still beats the collective execution time of around 300 ms of the validation set by one order of magnitude. And this comparison shifts only further in favour of the predictor model when

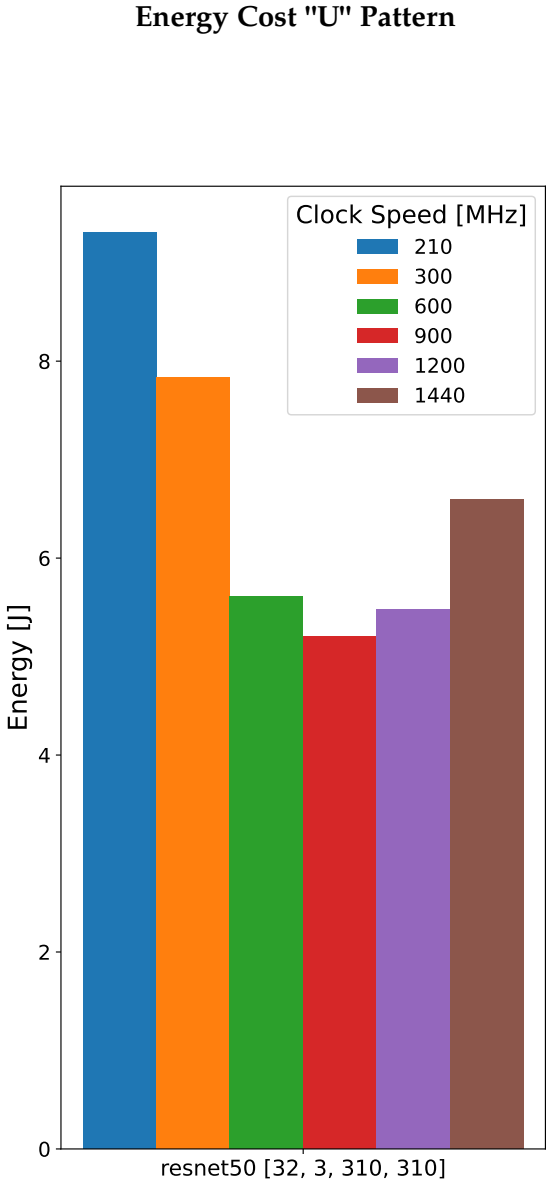

Figure 5.17: By showing the energy measurements across different clock speeds for only one model, the "U" pattern which is present for all models, becomes even more apparent. In terms of energy cost it is most efficient to run both inference and training tasks at a clock speed of 900 MHz for all our tested models.

we look at a still very reasonable scenario of training at a clock speed of 930 MHz. With this the collective execution time is close to 2000 ms, while the predictor time is not affected. Here we have an predictor latency over 60 times lower than performing the measurement instead.

6 DISCUSSION AND OUTLOOK

6.1 DISCUSSION

The approach we took in this work was of an exploratory nature. Rather than diving deep into one specific configuration, we chose to go wide and look at multiple hardware and many, many neural network scenarios. That approach turned out to be both blessing and curse. A blessing, because bare any limitations, the results would be applicable in a very wide field of applications and our findings are understandable at a more abstract level, which does not require a detailed understanding of subject matter. A curse, because we have to hamper our scientific curiosity not to follow every rabbit hole we encounter on our journey. Given the wide approach, we also encounter a wide range is troubleshooting issues, of various importance to the study itself. This balancing act of weighing the cost and benefit of following up interesting trends and deciding whether to invest the time to overcome obstacles was the most difficult part of this work. In the same way in which there is no end in going deep, no state of completeness, there is also no completeness to be achieved in going wide. And while, from an approriate distance, this is obvious, it is also fundamentally frustrating to deal with.
In order to go wide, one would like to include a large number of hardware scenarios and study all of them with the same set of accurate tools. In an ideal setting, we would include GPUs from Nvidia, AMD, Intel and further FPGAs and other ML accelerating coprocessors such as IPUs. But the platforms and tools are to varied, that even if we found a tool which was able to measure power for many of them, we would likely give up both time and power resolution in exchange for the improved compatibility. Another simple but very important limitation is the question of the hardware available to us. Because of these considerations as well as in order to keep the scope of this thesis in check, we decided to limit ourselves to Nvidia GPUs.
Another dimension to go wide in is the plurality of neural networks to include in our study. A central contribution to that decision was our platform of choice. Due to prior experience with the platform we chose PyTorch. In order to use well known networks and implementations, we chose to work with neural networks from the torchvision library. This also improves the easy of reproducing our study. However, our GPU with the smallest amount of global memory added a limit to which model-input sets we were able to include in our suite of model-input sets profiled for the training set. Each model-input set had to

complete all benchmarks on all hardware configurations. In a few cases a model-input set worked on the A30 but not on the 2080TI. The reason was not apparent in every case, and while in a larger work it might have been possible to determine it, this would have gone beyond the scope we were able to maintain here.
Our choices to study both time and energy for NN inference and training were shaped by a desire to study a novel section of this field, adding value to our research.
After initial attempts to include the clock speed study for both GPUs, after running into issues on the 2080TI, it was dropped from this part of the overall study.
All of this shapes into our scope. A scope which was not predetermined from the get go. Rather it evolved hand in hand with the study.
In an earlier phase of the study, the main validation of our measurement methodology took place. At that time, we had not yet began setting the clock speed. Instead we worked with both the 2080TI's and the A30's default configuration and added on top, the A30, but with disabled tensor cores.

THIS COVERS WHY WE TOOK THE PART OF THE MAP WE DID. NOW WE DISCUSS PREDICTION MODELS SHORTLY AND TRANSITION INTO VALIDATION AND ITS PURPOSE HERE THEN FOLLOWING UP WITH THIS WORKS APPLICATION. THIS SHOUD LEAD INTO THE OUTLOOK

6.2 OUTLOOK

6.3 CONCLUSION

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