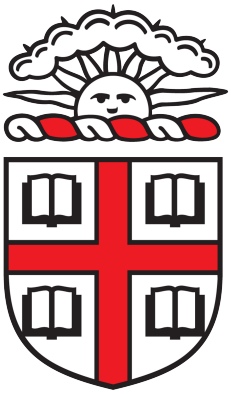
BROWN UNIVERSITY



UNDERGRADUATE HONORS THESIS

Efficient Multi-Task Learning for Augmented Reality

Author:
Austin FUNK

Advisor and First Reader:
Sherief REDA
Second Reader:
James TOMPKIN

A thesis submitted in fulfillment of the requirements for Honors in Computer Engineering

in the

Scalable Energy-Efficient Computing Laboratory (SCALE)
School of Engineering

April 28, 2025

Last edited October 8, 2025

BROWN UNIVERSITY

Abstract

Efficient Multi-Task Learning for Augmented Reality

by Austin FUNK

Augmented reality (AR) is a rapidly growing technology with a wide variety of applications, from medical research to entertainment. AR is generally governed by a single input from a camera, especially on low resource devices. In contrast, there is a wide range of potential tasks that often want to be performed in tandem, including semantic segmentation, normal estimation, and depth mapping. Multi-task learning (MTL) models provide a viable solution in contexts with notable compute constraints by replacing several models with one. Despite this, current MTL models struggle to perform inference quickly on mobile and edge devices, severely limiting applications that rely on real-time input data. Improving the speed of prediction on edge devices would increase the breadth and depth of current AR applications in several important fields, allowing for the use of constant video input in many parallel tasks.

This thesis presents a characterization study and novel, MTL-specific innovations. The study identifies two types of encoder backbones, convolutional neural networks (CNN) and transformers, and benchmarks several different architectures within each category, identifying broad bottlenecks and drawbacks for each type. This analysis motivates a series of experiments and methodologies that work in tandem for significant speedups, including multi-processor utilization, split model hardware scheduling, and inference pipelining. Each methodology and combination was benchmarked using real time input, providing important insight into practical engineering strategies. Multi-processor scheduling resulted in a maximum FPS improvement of 28.2% over its comparable CPU-bound model. With both scheduling and pipelining, FPS increased by approximately 79.3% over the baseline.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Interest in augmented reality (AR) has only grown in recent years. The number of papers related to AR[1] is approximately 932,000, with approximately 225,000 of them published since 2021. Even greater interest has been seen in deep learning, especially large language models (LLMs) and generative AI. "Attention is All You Need" [40], which introduced transformer-based models as a viable and more promising replacement for traditional convolutional neural networks, has over 160,000 citations as of this writing. The overlap between the two sectors has tremendous potential, including in medicine [15], architecture [10], and entertainment [33]. The primary roadblock is resource management; ResNet-50 [20], a convolutional encoder that accepts images as input, has over 25 million floating point weights. The weights alone conservatively require over 100 megabytes of storage, and this is for a model from 2015. Transformer based architectures can have tens of billions of weights; GPT-3 [3], has around 175 billion.

Potentially more important than the storage cost is the latency of any potential architecture. Using the example of GPT-3, the response time from a prompt is not as important as its accuracy. Given a complex prompt or the expectation of a complex answer, most companies and researchers have determined that greater latency is reasonable, as shown by the latest "Reasoning" options that many LLMs now provide. In contrast, augmented and virtual reality contexts are very sensitive to latency. If a visual output is expected to replace the original camera input, the rate at which output frames are produced must be fast enough to avoid motion sickness. This caveat also severely limits the batch size of input frames, considering that every frame should be as evenly temporally spaced as possible to minimize user discomfort. Passing an input tensor, say of dimension $224 \times 224 \times 3$ in the case of ResNet, through tens of layers of computationally expensive multiplications is therefore prohibitive in low latency, low resource applications. This bottleneck is even more defined when analyzing transformer-based architectures, which are newer, bigger, and more common in current research. This thesis, therefore, focuses primarily on improving runtime for hardware running transformer-based architectures, specifically using the Swin transformer [28] developed by Microsoft.

This leads us to the introduction of the two foci of this thesis. The first is a characterization study of machine learning models on mobile and edge devices using a newly developed Android application called **BeAR**, or **Be**nchmarking in **A**ugmented

[1] per Google Scholar

Reality. BeAR was developed specifically for real time, real life reporting of how various multi-task learning (MTL) models interact with edge and mobile devices. This study examines the benefits of MTL models over traditional models for edge-based AR applications, and how different baseline MTL models perform in real scenarios. This data is then used to identify bottlenecks in processing, leading to minor application speedups and novel approaches to performing inference with MTL models for AR.

The second focus of this thesis is on the methodologies that were experimented with, including dynamic processor selection, manual processor and thread assignment, hardware scheduling models using a splitting technique unique to MTL, and a combination of scheduling and pipelining for parallelism and significant improvements over sequentially executed inference. Analysis shows that the combination of these techniques with efficient MTL models produces promising real-world runtime improvements over naïve, hardware and/or model agnostic execution. Multiprocessor scheduling results in a maximum FPS improvement of 28.2% over its comparable CPU-bound model. With both scheduling and pipelining, FPS increases by approximately 79.3% over the baseline.

Chapter 2

Background

2.1 Modern Deep Learning

Deep learning and computer vision have been intertwined for a long time in the context of computer science research [20] [28] [27]. With deep learning, the goal is to write a program to find a set of equations and values that can produce correct outputs given unknown inputs. The definition of correct, and what those inputs and outputs look like, is dependent on context. LLMs are a topical example of deep learning, where the input and output are both text, though recent developments in multimodality have challenged this notion [18].

A more relevant example is centered around the MNIST dataset, a seminal series of 70,000 labeled 28×28 images of handwritten digits between 0 and 9, inclusive [27]. The motive behind creating this dataset was to write a program that could correctly label a handwritten digit with a high level of accuracy, an effort by the US Postal Service to help read zip codes. This context, then, is one where images of a specific size are expected to be identified and classified based on common characteristics, which would be a near-impossible task to complete manually. The MNIST challenge would cause the development of the first well-known and effective convolutional neural networks (CNNs). More details and their relevance to this work will be discussed in section 2.1.1.

Transformer-based architectures have been at the forefront of machine learning research since Google Brain's "Attention is All You Need" [40] in 2017. Attention in deep learning is a method of relating distant information, such as a word at the beginning of a sentence to a word at the end of the sentence. While attention wasn't a new development [34] [41] [2], Vaswani *et al.* were the first to suggest that convolutional and recurrent layers weren't necessary. Today, there are several successful applications of transformers for vision tasks, including Swin [28], one of the models used in this thesis.

2.1.1 Convolutional Neural Networks (CNN)

The first CNN was developed for the 1989 paper "Handwritten Digit Recognition with a Back-Propagation Network" [27], though CNNs wouldn't take off until "ImageNet Classification with Deep Convolutional Neural Networks" [26], colloquially known as AlexNet. CNNs work by gathering local information in images via convolution, or cross-correlation in the world of signal processing.

$$G(i,j) = \sum_{u=-k}^{k} \sum_{v=-m}^{m} w(u,v) \times I(i+u, j+v) \tag{2.1}$$

FIGURE 2.1: Cross correlation for discrete signals.

Tensors of learnable weights are "dragged" across labeled training data (see figure 2.2)[1], encoding the information into a high dimensional space that can be used for classification, or upscaled with more convolutions into a new image. Researchers found that when these image convolutions are placed in sequence, they are able to get remarkably good results. AlexNet proved that deep-layered networks were not only feasible, but better than the competition, achieving sub-25% error for the first time in the ImageNet challenge's history [26]. CNN research would dominate for the next five years or so, peaking with ResNet [20] with a maximum of 152 layers compared to AlexNet's 5.

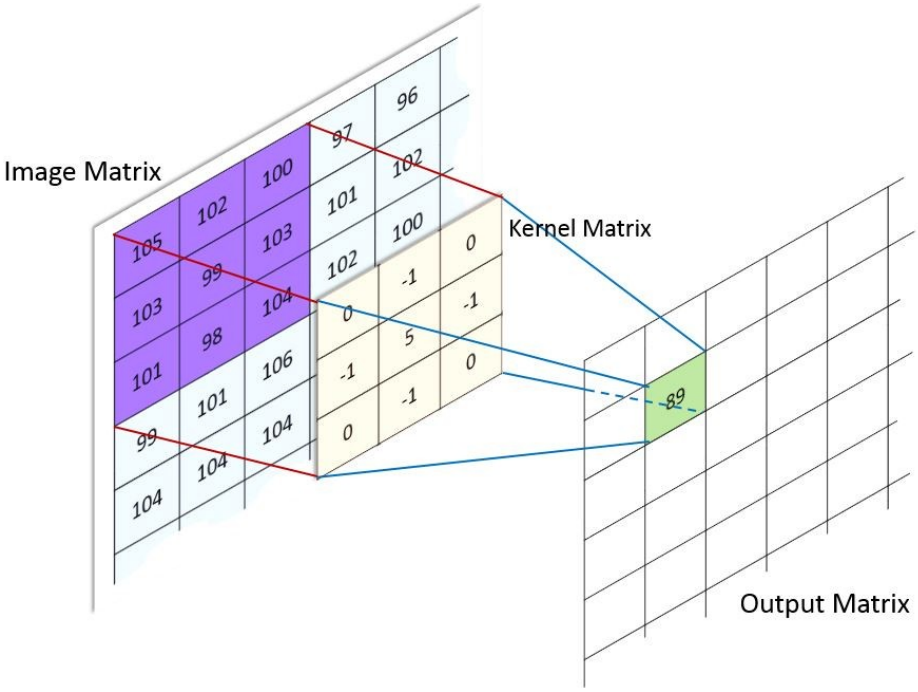


FIGURE 2.2: Visualization of image convolution [23].

2.1.2 Transformers

Transformers were initially designed for text-to-text applications, however in the original paper the authors acknowledge the potential for vision applications [40]. Scaled dot-product attention, the methodology for attention described in Vaswani *et al.*, uses the equation in figure 2.3 to determine learnable weights. It can be understood as a

[1] w is the weight kernel, k and m are half the height and width of the weight kernel respectively, I is the original image, and $G(i,j)$ is the resultant pixel at (i,j). Often, k and m will be equal.

$$\text{Attention}(Q, K, V) = \text{Softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2.2}$$

FIGURE 2.3: Scaled Dot-Product Attention [40].

search function, where a query Q is applied to a series of key-value pairs K and V with the goal of finding weights that correctly predict values. Multi-head attention extends scaled dot-product attention to incorporate previous layers, eliminating the issue of disappearing gradients with previous recursive neural networks (RNNs) without using subpar, non-parallelizable methods for layer memory [21].

A relevant aspect of the Transformer architecture is the use of an encoder and decoder, as seen in figure 2.4. The encoder is shown on the left accepting the current inputs while the decoder is on the right accepting all previous outputs[2]. The impact of this clear split in the model architecture is explored further in section 2.2.

Swin [28] is a Transformer backbone published in 2021 that aimed to solve one

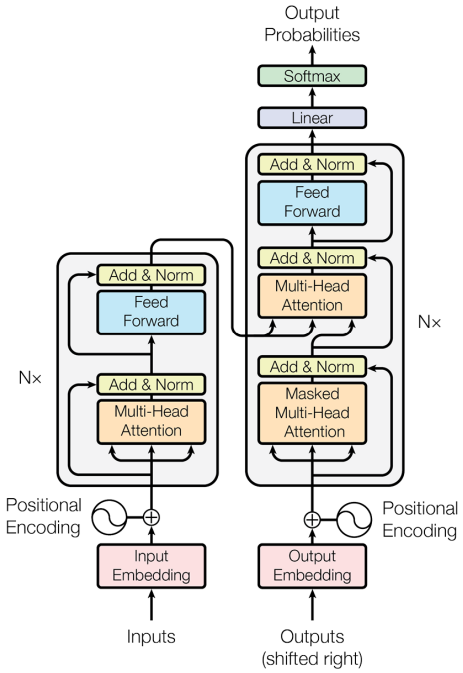


FIGURE 2.4: Transformer architecture [40].

of the biggest problems with applying Transformers to vision tasks: Images contain far more information than text prompts, which makes it difficult to apply visual input directly, due primarily to insurmountable training times. Swin tackles this by intellectually borrowing from CNNs; images are divided into larger segments, then a shifting window groups these segments[3] for self-attention. Figure 2.5 shows a comparison between Swin and ViT [14], a predecessor that didn't shift windows or vary window size across layers.

[2]The outputs are shifted by one to verify that the output at i only depends on outputs from positions less than or equal to $i - 1$, i.e. only the known outputs.

[3]Unlike ViT [14], window sizes are dependent on the layer.

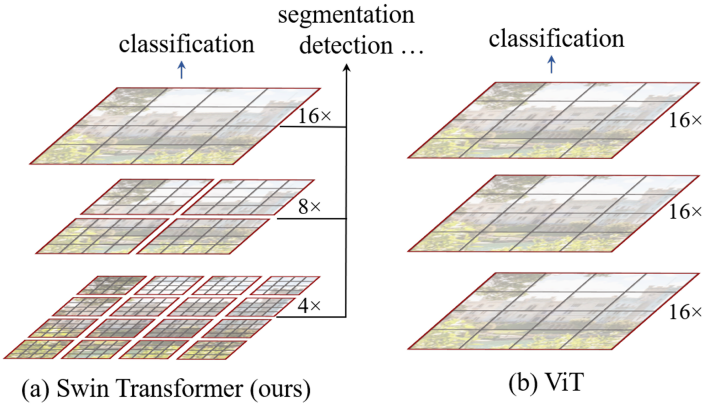


FIGURE 2.5: A strategy comparison between Swin and ViT.

2.2 Multi-Task Learning (MTL)

Most innovations in deep learning have been agnostic to any metrics except accuracy and inference time [39] [36]. While both of these metrics are important, things like power and temperature become increasingly relevant in low resource environments, such as with mobile and edge devices. MTL is a promising way to balance fewer resources with acceptable accuracy.

The term "multi-task learning" [5] was first used in 1997, defining the methodology as one that learns multiple tasks in tandem, allowing them to influence each other in a shared representation space. This allows inference over multiple tasks with a single model, which exactly fits into the problem statement of inference for AR: One image needs to produce multiple outputs for minimal energy, space, and time. In many modern applications, this takes the form of an encoder-decoder model similar to the Transformer [40], with the distinction that MTL models utilize multiple decoders for separate tasks, as seen in figure 2.6. Relevant applications of MTL models are discussed in sections 2.4.1 and 2.4.2.

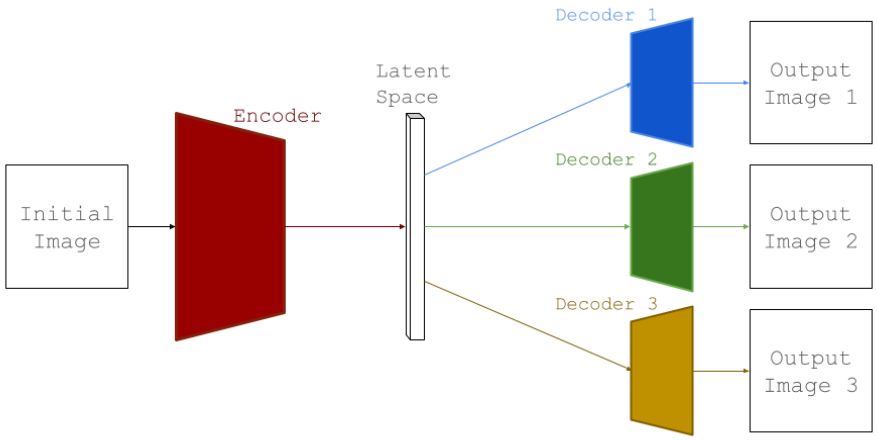


FIGURE 2.6: A high-level layout of MTL execution.

2.3 Scheduling

Modern computers, including many mobile and edge devices, have both multiple processors and multiple *types* of processors. The idea behind scheduling is to optimize hardware and its usage for specific categories of tasks, as opposed to or in tandem with the optimization of software algorithms. It is most often used in high efficiency systems, including real-time mobile and edge applications.

Scheduling can be applied at all layers of the hardware stack. At the PE[4] level, scheduling solutions utilize different memory units to maximize data reuse and limit data movement. An example strategy is SOC-MOP [7], which stands for "(**s**)ingle (**o**)utput feature map (**c**)hannel - (**m**)ultiple (**o**)utput feature map plane (**p**)ixels". This strategy, then, maximizes pixel reuse over a single channel in the output image, which is ideal for convolutions. MOC-SOP, which stands for "(**m**)ultiple (**o**)utput feature map (**c**)hannel - (**s**)ingle (**o**)utput feature map plane (**p**)ixel", is optimized for reuse over a single pixel with many channels, making it ideal for fully connected layers. There are many other strategies that try to strike a middle ground, or attempt reuse across all maps (such as Eyeriss).

```
# P = Output activation height, Q = Output activation width
# M = Output channels, C = Input channels
# R = Filter height, S = Filter width
# N = Batch size, Z = Output, I = Input, W = Filter
for p1 in range(P1):
    for q1 in range(Q1):
        for m1 in range(M1):
            for c1 in range(C1):
                parallel_for p0 in range(P0):
                    parallel_for q0 in range(Q0):
                        for r in range(R):
                            for s in range(S):
                                for n in range(N):
                                    for m0 in range(M0):
                                        for c0 in range(C0):
                                            Z += I * W
```

FIGURE 2.7: Example loop nest for SOC-MOP.

This method of scheduling is lower level than this work goes since the purpose is not to design new hardware, but to optimally use existing hardware. Some of these strategies, such as loop reordering and tiling, will be discussed again briefly in section 4.1.1. Scheduling, as far as this work is concerned with it, will occur at the processor level, specifically how best to distribute workloads over space and time on a given system. Experimentation in later sections takes into account an NPU and CPU, though the discoveries and strategies can easily be applied to other types of processors, such as GPUs or TPUs.

[4]Processing Element.

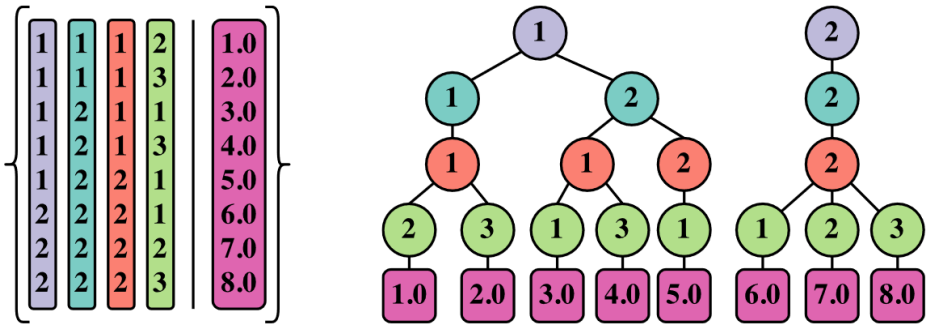
2.4 Machine Learning in Relevant Applications

The problem this thesis wishes to address – that of bringing the capabilities of modern machine learning to low resource environments – is not new or unexplored. An understanding of previous work is essential for understanding the context and novelty of this thesis.

2.4.1 Mobile

Bringing deep learning to mobile or low resource environments is an open research question, though not without progress. Some general methods for faster inference and lower memory requirements include quantization, sparsity, and pruning [4].

- Quantization replaces 32 bit floating point values with smaller representations in the model's weights, such as 16 bit floating point values, 8 bit integers, or even 4 bit integers [12] [43]. This can occur in the training stage [17], but is more often employed post-training [44].
- Sparsity takes advantage of predictable multiplications, most often by zero, by replacing classic sequential representations with sparse data structures. One way to do this is with a sparse tree, where each tree layer could be a tensor dimension in a simple case, or the relationships between nodes can be far more complex and optimized [19].
- Pruning is the process of removing features from a model that contribute nothing or very little to the accuracy of the output, though determining exactly which features can be removed is extremely difficult [8].



Coordinate format. CSF: conceptual.

FIGURE 2.8: A conceptual representation of Compressed Sparse Fiber [37], one way to efficiently represent and store a sparse matrix.

Previous hardware restrictions made learning and inference on mobile devices especially challenging, but recent research has brought about NPUs, which are processors designed for neural networks [22]. NPUs are becoming increasingly important in the age of AI, with over 189,000 papers on the topic[5]. While papers exist with detailed, low-level benchmarks [29] [38], few results exist for real-time data with practical system overhead, which is important for broad application.

[5] per Google Scholar.

2.4.2 Augmented Reality

The application of deep learning to AR exists in several major fields [10] [33] [15]. Many solutions attempt to evade hardware deficiencies by utilizing the cloud [13], however the transfer of data wirelessly can be computationally expensive and increase latency, while also being limited by the availability of a stable internet connection.

There is relatively little research related to multi-task learning in augmented reality applications. AdaMTL [31], developed in SCALE lab, is a framework that learns alongside the model to identify which tasks require which layers, then adaptively activate only the necessary layers at inference time. This allows for a significant reduction in latency, energy use, and computational complexity when compared to the same model without the framework. This is a training-focused approach to improving edge-based inference for AR, compared to this work's focus on pretrained MTL models and hardware utilization.

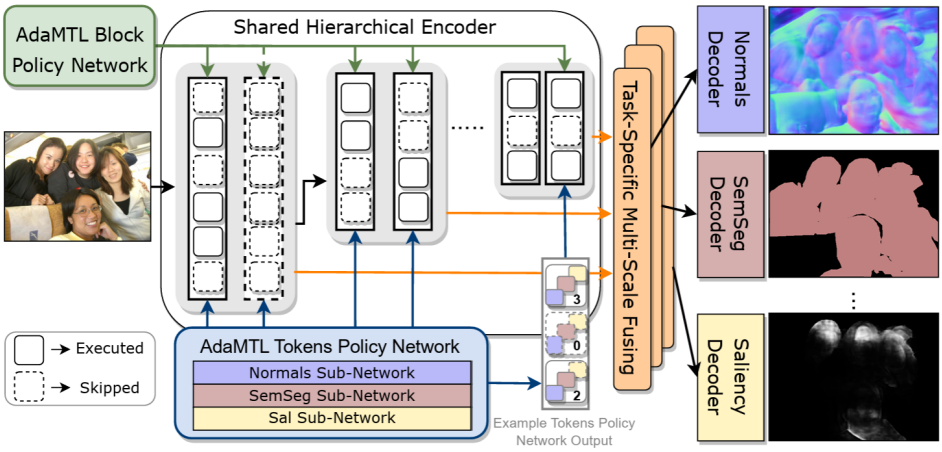


FIGURE 2.9: A visualization of how AdaMTL interacts with a model at inference time [31].

Another relevant example are models that learn on egocentric data [25]. Kapidis *et al.* applied multi-task learning to egocentric data because they wanted to allow different tasks to influence each other, which resulted in an increase in object detection accuracy over other state-of-the-art models. This is further evidence that MTL models are preferable to single-task models for augmented reality; not only does MTL have lower resource requirements, but it can match or increase performance compared to traditional models.

Chapter 3

Characterization of Vision Models

There has been significant research surrounding profiling different vision architectures [39] [36], but relatively few that focus more directly on how these architectures respond to real devices with limited hardware. Benchmarking multiple models – ResNet and Swin – across several architecture types on two different devices helps to reveal realistic expectations and performance bottlenecks to be addressed. The devices used for this study were a Samsung Galaxy S24 smartphone (figure 3.2a) and the Vuzix M4000 AR glasses (figure 3.2b), identified by their respective processors in relevant tables.

3.1 Modeling Performance

The primary goals of this study were to determine:

1. How do modern CNNs compare to Transformers?
2. What bottlenecks exist and how can they be addressed?

Some initial metrics were recorded before postprocessing improvements were made, hence why some relevant metrics are excluded; such cases are clearly labeled.

3.1.1 Benchmarking in Augmented Reality (BeAR)

A benchmarking application for AR on mobile devices was previously developed by SCALE lab under the name ARBench [9]. Due to limited customizability, device incompatability, and a desire for metrics based on real-time camera data, **Be**nchmarking in **A**ugmented **R**eality (BeAR) was developed. BeAR is specifically designed for MTL models, giving researchers an extensive suite of metrics recording options, multiple input methods, and all of the experimental execution methods discussed in chapter 4. BeAR also supports an unlimited number of model outputs, making it ideal for MTL. Figure 3.1 shows a version of the app on the Vuzix M4000[1]. The modular design of the app allows for easy additions, including in output methods, settings, and model availability. All of the data collected for this thesis was done using BeAR, and is hosted on GitHub[2] along with a detailed README.

[1]https://www.vuzix.com/products/m4000-smart-glasses
[2]https://github.com/scale-lab/BeAR-app

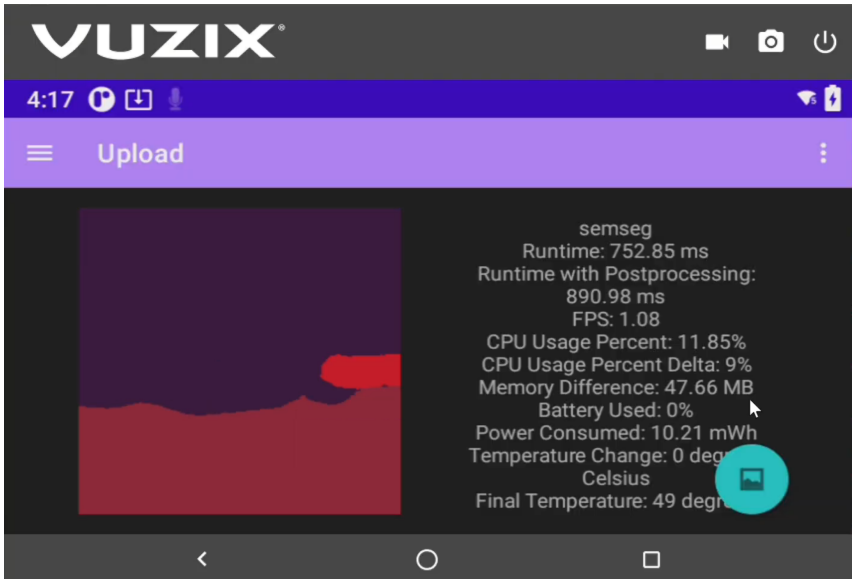


FIGURE 3.1: Testing BeAR on the Vuzix M4000.

Image Processing

BeAR accepts image inputs via device storage (for single images) or via the device's camera (for image streams, i.e. video). Images go through minimal preprocessing, only a format change and dimension change if necessary. Output data can be postprocessed in a variety of ways as specified by the user. Most depth processing is done by converting a single channel intensity output to RGB, resulting in a black and white heat map. Segmentation processing for all metrics is done with a color map, where a set number of colors are predetermined to decrease compute cost. There is also an option to convert colors over a gradient for outputs with many channels. Image processing is done quickly by utilizing JNI, a programming interface that allows Java applications to run C++ code.

Software Hierarchy

BeAR centralizes inference to a single monitoring object, which is stored inside a controller object that's accessible to the UI. Scheduling (seen in section 4.1) is done by having the user specify an encoder file and any number of compatible decoders, then creating different runtime sessions for each that specifies which processor to use. Pipelining (seen in section 4.2) takes the infrastructure of scheduling and launches concurrent threads, using a synchronized queue for inter-process communication.

3.1.2 Relevant Metrics

Determining which metrics to give attention to can be determined by looking at the approaches of previous work; AdaMTL [31] accounts for energy usage and inference time, and eAR [13] focuses primarily on energy consumption to increase battery life. Due to the usage of primarily preexisting, pretrained models, and the desire for real-time camera input for benchmarking, model accuracy is not directly accounted for.

(A) The S24 from the AT&T product page.

(B) The Vuzix M4000 from the product page.

FIGURE 3.2: Devices used for benchmarking.

When relevant, such as in section 4.1 when using pretrained models becomes less feasible, accuracy metrics are reported. However, the purpose of this study is to determine how to optimize hardware utilization, not how to optimize the model itself[3].

Therefore, the metrics recorded by BeAR for analysis are reported in table 3.1 along with their relevant units. Any important notes, such as the reliability of each metric, can also be found in this table.

Metric	**Units**	**Notes**
Inference Time	Milliseconds (ms)	N/A
Inference + Postprocessing	Milliseconds (ms)	N/A
Frames per Second	FPS	N/A
CPU Usage	Percent Difference	Unreliable due to interfering processes
Memory Usage	Megabytes	Unreliable due to interfering processes
Battery Usage	Percent Difference	Requires $>$ 1s of runtime
Energy Consumption	Microwatt-Hours	Requires $>$ 1s of runtime
Current	Microamps	Requires $>$ 1s of runtime
Temperature Delta	Degrees Celsius	N/A
Final Temperature	Degrees Celsius	N/A

TABLE 3.1: Recorded hardware metrics for characterization.

[3]Quantization is used at various points for benchmarking due to parallel research from Harb Lin (shangran_lin@brown.edu).

Device	Inference Time (ms)	Frames per Second
Snapdragon 8 Gen 3 (S24)	136.35	4.38
8 Core Qualcomm XR1 (M4000)	502.83	1.82

TABLE 3.2: Runtime differences with ResNet-50 between devices.

Device	Energy Consumption (mWh)	Temperature Delta (°C)	Final Temperature (°C)
Snapdragon 8 Gen 3 (S24)	0.43	0.30	28.2
8 Core Qualcomm XR1 (M4000)	22.53	0.33	32.67

TABLE 3.3: Energy and temperature differences with ResNet-50 between devices.

3.1.3 CNN Performance

The primary CNN used pre-optimization was ResNet-50 due to its popularity and balance between accuracy and size. After changes to postprocessing, ResNet-18 was also tested to compare both models with minimal interference from other internal processes. Both models were pretrained from MTLKit [1] with image dimensions of $(224, 224)$ and two tasks, normals detection and segmentation. These tasks differ from the tasks used for benchmarking the Swin-based models in section 3.1.4 due to the number of output channels available in different datasets and a need to maintain consistency across transformer models, specifically with respect to the dataset chosen for the quantized Swin-based model.

Table 3.2 compares average inference time and FPS of ResNet-50 over three 10 second[4] trials on both devices. Table 3.3 compares temperature and energy data over those same trials. The S24 outperforms the M4000 by a significant margin in runtime, FPS, and power usage over time, but comparable in terms of temperature difference.

Table 3.4 compares average inference time, runtime with postprocessing, and FPS for ResNet-50 and ResNet-18 on the S24 over ten trials, each 10 seconds in length. Table 3.5 contains energy consumption and temperature data for this same set of trials. The boost in performance for ResNet-50 is due to better postprocessing, frame throttling, and allowing basic ONNX runtime optimizations. ResNet-18 outperforms ResNet-50 in 4 out of 5 metrics, falling short only in average temperature increases, though these metrics don't account for accuracy.

The average temperature delta for both models is concerning; assuming a linear increase in temperature, it would take ResNet-18 just under 2 minutes to raise the device's temperature 10°C, and ResNet-50 just under 5 minutes for the same temperature increase. Long term usage is therefore infeasible without extreme temperature control. Attempts to control temperature are discussed in section 4.1.1.

[4]Longer trials were shown to have very similar results due to diminishing returns, i.e. the number of frames processed after 10 seconds is great enough to gather valid data.

Model	Inference Time (ms)	Runtime (ms)	Frames per Second
ResNet-18	73.08	77.34	10.62
ResNet-50	110.17	114.75	6.97

TABLE 3.4: Runtime differences between ResNet-50 and ResNet-18.

Model	Energy Consumption (mWh)	Temperature Delta (°C)
ResNet-18	0.23	0.84
ResNet-50	0.27	0.34

TABLE 3.5: Energy and temperature differences between ResNet-50 and ResNet-18.

The runtime for ResNet-18 is over 52% faster than ResNet-50 on average, at a significant decrease in accuracy in the context of machine learning. For classification, ResNet-18 has a top-1 accuracy of 69.6% and ResNet-50 has a top-1 accuracy of 76.9% [35]. For segmentation specifically on the Cityscapes dataset [11], ResNet-18 has an MIoU[5] of approximately 60.0% and ResNet-50 has an MIoU of approximately 71.2% [32]. Therefore, the primary bottleneck for ResNet models is temperature, and different layer counts provide a tradeoff between accuracy and runtime.

3.1.4 Transformer Performance

Swin style encoders were chosen for benchmarking due to the abundance of versions, parallel research being done in SCALE lab on quantizing Swin models, and high relative accuracy compared to other image-based transformers [28]. Swin has been developed in various sizes, specifically Tiny, Small, Base, and Large, with each having progressively more parameters and slight changes to model architecture[6]. Swin encoders can be paired with a large variety of different tasks; the two tasks chosen for benchmarking were depth[7] and segmentation due to their relevance to AR applications and availability of training data via NYUv2D [30]. Combining the four encoders with the same depth and segmentation decoders gives us a collection of models we call **SwinDS-#**, where the **#** is replaced by the identifying initial of the encoder[8]. A quantized[9] version of SwinDS-T is referred to as SwinDS-Q throughout this work. Tables 3.6 and 3.7 compare four models, excluding SwinDS-L due to its infeasibility in mobile applications, using the same metrics and methods seen in section 3.1.3 using the Snapdragon 8 Gen 3 (S24).

The comparison between the various SwinDS iterations is clear; SwinDS-T has the highest FPS and lowest energy consumption compared to its other unoptimized

[5]Mean Intersection over Union is a common metric for evaluating models with image outputs.
[6]A few examples include layer count and embedding size.
[7]Sometimes referred to as "saliency" in literature [31].
[8]For example, the model SwinDS-B uses the Base encoder.
[9]The model was quantized to int8 precision by Harb Lin.

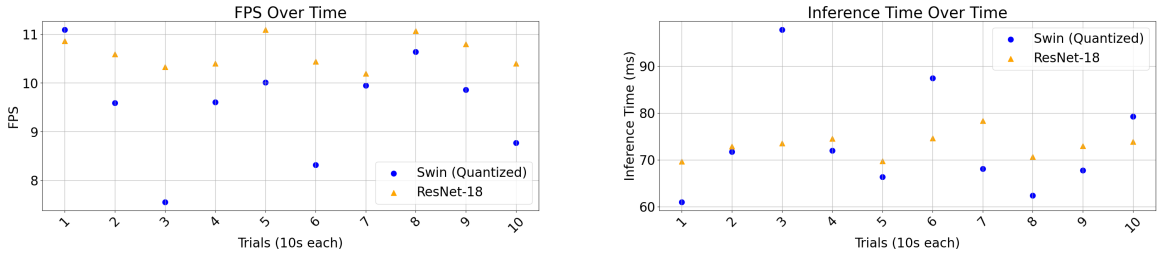
Model	Inference Time (ms)	Runtime (ms)	Frames per Second
SwinDS-B	319.34	323.89	2.95
SwinDS-S	196.96	201.79	4.59
SwinDS-T	113.03	117.49	7.53
SwinDS-Q	73.37	86.95	9.54

TABLE 3.6: Runtime differences between SwinDS models.

Model	Energy Consumption (mWh)	Temperature Delta (°C)
SwinDS-B	1.03	0.23
SwinDS-S	0.55	0.35
SwinDS-T	0.32	0.32
SwinDS-Q	0.22	0.02

TABLE 3.7: Energy and temperature differences between SwinDS models.

counterparts at the expense of accuracy, and quantizing it with int8 precision results in further improvements, again at the expense of accuracy. The more interesting comparison is with the convolutional models. ResNet-18 and SwinDS-Q have nearly identical average inference times, yet SwinDS-Q ends up with a lower FPS due to a more dramatic postprocessing step. ResNet-18 was trained using the Pascal VOC dataset [16] expecting 8 channels for segmentation, while SwinDS-Q was trained with the NYUv2 dataset [30] expecting 20 channels for segmentation. Figure 3.3 shows two plots, one for inference time and one for FPS, comparing ResNet-18 and SwinDS-Q.



(A) Average FPS over each trial.

(B) Average inference time over each trial.

FIGURE 3.3: Plotted comparison of the fastest SwinDS variant versus the fastest CNN variant tested.

Considering that inference time is nearly equivalent for ResNet-18 and SwinDS-Q, SwinDS-Q is preferred due to its slightly lower energy consumption, significantly lower temperature delta, and more modern architecture. It should be noted that ResNet-18 isn't quantized, so there is still some potential speedup to be had, though the accuracy decrease might be prohibitive.

The data presented cannot be used to make broad claims about all CNNs or all Transformers. However, both ResNet and Swin architectures and prominent, high

performing backbones in each of their respective categories, therefore making them reasonable baselines to design around. The primary bottleneck for both backbones is inference time. Studies show that in non-AR contexts, the minimum recommended FPS is 15 for interactive tasks [6]. More recent studies suggest that much higher frame rates are necessary to prevent motion sickness in virtual reality, upwards of 120 FPS [42]. ResNet-18, the highest performing model in terms of FPS, runs at under 11 FPS, far from even the minimum for interactive tasks. Optimizations discussed in chapter 4 must therefore focus on inference time for the sake of viability, but other metrics should still be considered, primarily energy and temperature.

Chapter 4

Hardware-Software Co-design

Based on characterization metrics and the corresponding analysis, we propose methods focusing on full hardware utilization, primarily looking to improve FPS without sacrificing energy or temperature. The methods also looked to utilize the unique architecture of MTL models, using scheduling, pipelining, and multithreading to take full advantage of the available hardware and flexibility of exchangeable decoders.

4.1 Hardware scheduling for MTL Models

Using different processors, especially those optimized for machine learning[1], is expected. First, we will look at baseline scheduling metrics utilizing the Samsung Galaxy S24's neural processing unit (NPU), then take advantage of the natural division in MTL models to spread the load of inference across processors.

4.1.1 Naïve Scheduling on the NPU

Simply running the same models on the NPU instead of the CPU results in the same or worse performance. Tables 4.1 and 4.2 report metrics utilizing only the S24's NPU over 10 trials of 10 seconds each, averaged. SwinDS-Q is excluded due to incompatibilities with NNAPI[2], which is necessary for NPU utilization on Android devices.

Compared to tables 3.4, 3.5, 3.6, and 3.7, ResNet-18 experienced an approximate 10.0% drop in performance across time-based metrics, an 8.7% increase in energy usage, and a 34.5% drop in average change in temperature; ResNet-50 experienced a

[1]Examples include programmable GPUs and Google's TPU [24], which is more similar to an ASIC.

[2]Documentation can be found here: https://onnxruntime.ai/docs/execution-providers/NNAPI-ExecutionProvider.html.

Model	Inference Time (ms)	Runtime (ms)	Frames per Second
ResNet-18	81.08	85.87	9.65
ResNet-50	129.31	135.07	6.55
SwinDS-T	109.71	114.33	7.70

TABLE 4.1: Time metrics for the best performing CPU models when run on the NPU.

Model	Energy Consumption (mWh)	Temperature Delta (°C)
ResNet-18	0.25	0.55
ResNet-50	0.40	0.82
SwinDS-T	0.36	0.83

TABLE 4.2: Energy and temperature metrics for the best performing CPU models when run on the NPU.

Loop Order	Thread Count	Tile Size	Postprocessing Time (ms)
H→W→C	1	Full	50.1
W→H→C	1	Full	51.1
H→W→C	1	H/2, W/2	52.4
H→W→C	2, over W0	H/2, W/2	51.8
H→W→C	1	H/4, W/4	51.8
H→W→C	2, over W0	H/4, W/4	53.1
H→W→C	2, over W	Full	49.9
W→H→C	2, over H	Full	54.0
H→W→C	4, over W	Full	64.5

TABLE 4.3: Metrics from different loop orders and tiling strategies.

14.8% drop in execution time, a 6.4% drop in average frames per second, a 48.1% increase in energy usage, and a 141.2% increase in average change in temperature; SwinDS-T experienced a 3.0% increase in execution time, a 2.3% increase in frames per second, a 12.5% increase in energy usage, and a whopping 405.0% increase in average change in temperature.

Despite minor improvements to runtime for SwinDS-T, utilizing the NPU without considering other methods of optimization is insufficient for faster frame rates, energy usage, and thermal concerns.

Tiling and Loop Reordering

A relatively small sample of experiments were conducted with various tiling and loop ordering strategies, specifically for postprocessing. Table 4.3 was recorded using ResNet-18. Note that the recorded postprocessing times are high due to other methods not being active in tandem.

Even with multithreading, it was found that any amount of tiling resulted in slower or equivalent postprocessing speeds, and that loop reordering was irrelevant. This was determined to be because of compiler optimizations and significant enough cache sizes such that tiling only added overhead without speedups and that loop reordering had no effect. Future experiments should focus on data stationary approaches for inference, such as output stationary or row stationary [7]. However, the abstraction seen in Java's NPU and GPU interactions make this a difficult task, and therefore may be better explored using tools other than BeAR.

Time Period (s)	0→10	10→20	20→30	30→40
Processor	NPU	NPU	CPU	CPU
FPS	6.83	7.32	4.22	4.40
Final Temperature (°C)	29.6	31.4	31.4	32.6
Temperature Difference (°C)	0.0	1.8	0.0	1.2

TABLE 4.4: Timeline of frame dip when throttling. Using SwinDS-T, Threshold manually set to 30.0°C.

Dynamic Processor Selection

One of the major concerns with deep learning on AR devices is overheating, considering the proximity of the device to users. This presents more severe safety hazards when compared to deep learning on other device types. One of the more conventional methods for temperature control is system throttling, which has the negative effect of drastically increased latency. A better approach is to find a middle ground between throttling and overheating, finding aspects of a system that can be dialed down when thermals climb beyond safe thresholds.

We experimented with dynamic processor selection. The idea behind this strategy is to alternate between available processors anytime one of them reaches an unsafe temperature. However, we found that this approach to be ineffective due to the NPU and CPU computing data at different speeds, which caused a severe dip in frame rate (see table 4.4). Another failure of this strategy was that, even when switching to the underutilized processor, overall system thermals continued to rise, albeit slightly slower than without processor switching. The tradeoff of slower thermal throttling for much worse throttling didn't make sense, and thus didn't make it into the final product. Future work could explore switching between models with the same class of backbone and outputs, such as between SwinDS-S and SwinDS-T, similar to "Play It Cool: Dynamic Shifting Prevents Thermal Throttling" [45]. This idea could also be extended to quantized models for decreased latency at the cost of accuracy.

4.1.2 Utilizing MTL Architectures for Scheduling

The division between the encoder and decoders, as well as there only being one expensive encoder for several computationally cheaper decoders, makes MTL models ideal for hardware scheduling. Specifically, splitting MTL models allows for more options in how hardware is allocated per section of the model. We experimented with different scheduling schemes, analyzing the effect of each permutation of the encoder/decoders on the CPU/NPU. We discovered significant frame rate improvements compared to the CPU-bound baseline based on the performance metrics discussed in chapter 3.

Division Drawbacks

There are two primary drawbacks with splitting models. The first is that the process of splitting a model and integrating it with an existing system can be nontrivial. While MTL models are easier to find split points in than other models due to the clear encoder-decoder structure, these splits are very rarely as simple as the one depicted in figure 2.6. For example, Swin encoders output four different resolutions of feature

Encoder Time (ms)	Encoder + Decoder Time (ms)	Frames per Second	Energy Consumption (mWh)	Temperature Delta (°C)
Encoder on NPU, Decoders on CPU				
105.15	204.92	4.27	0.13	0.39
Encoder on NPU, Decoders on NPU				
154.37	266.21	3.60	0.27	0.68
Encoder on CPU, Decoders on CPU				
189.95	285.69	3.33	0.35	0.59
Encoder on CPU, Decoders on NPU				
233.21	346.99	2.83	0.23	0.35

TABLE 4.5: Split SwinDS-T with four possible combinations for encoder/decoder scheduling.

maps to be used for various decoding tasks, which makes directing encoder outputs to decoder inputs difficult, depending on how each decoder in the system is implemented [28]. From a software perspective, it would be ideal for decoders to accept all four encoder outputs as input, but the effects this has on performance compared to only accepting relevant feature maps is untested. Moreover, determining exactly where to make a split in a model sometimes requires a deep understanding of the model's underlying architecture, which is not ideal for quick prototyping.

The second drawback is the increase in overhead, especially in Java applications. While a model split into its components should theoretically take up the same amount of memory as a whole model, new metadata and inference environments[3] for each component can easily balloon. The current design requires the encoder and decoders to each be separate files, which leads to $1 + n$ inference environments, where n is the number of decoders. Each inference environment can add tens of milliseconds to every loop, which is difficult to make up for, especially in real-time applications.

Sequential Execution Results

Utilizing the S24's CPU and NPU, there are four obvious scheduling permutations[4]. The model used for benchmarking has a SwinDS-T encoder and two custom decoders[5] due to limited pretrained decoders for this specific application. All data was averaged over ten seconds each, then averaged over ten trials. This should be considered a baseline for section 4.2.

Compared to previous trials in chapter 3, runtimes in table 4.5 are much greater, the worst being scheduling with the encoder on the CPU and decoders on the NPU. However, the scheduling case with all parts of the model on the CPU should be much closer to SwinDS-T in table 3.6, but instead we see a runtime increase of over 53%. Testing shows that overhead from multiple ONNX models does not cause the entirety of this effect, especially considering that encoder runtime on its own is greater than full

[3] See ONNX runtime documentation for more information.

[4] Other permutations would involve dividing the number of decoders in a model between processors.

[5] These decoders were constructed using PyTorch's prebuilt modules and trained on NYUD [30] for segmentation and depth.

model runtime in table 3.6 in all configurations except with the encoder on the NPU and decoders on the CPU. This suggests issues with the models rather than with the scheduling process itself. Therefore, a fairer comparison encompasses only split models, with CPU-CPU being the baseline.

NPU-CPU scheduling outperforms all other configurations in four out of five metrics, falling behind CPU-NPU in temperature delta by approximately 11.4%. CPU-NPU performs worse in all speed metrics and is squarely middling in energy consumption. NPU-NPU outperforms CPU-CPU by 8.1%, which is significantly greater than the whole model's equivalent metric of 2.3% in table 4.1. However, due to its dominance in four metrics and close second place in the last metric, NPU-CPU is the ideal choice for split Swin MTL models in most applications. NPU-CPU boasts a 28.2% increase in FPS over the baseline CPU-CPU configuration, though this may not hold true for CNNs or drastically different Transformer architectures.

4.2 Low Latency Pipelining for MTL Parallelism

To fully utilize the benefits of MTL-focused hardware scheduling, parallelizing the execution of model segments is essential. However, the amount of parallelism possible is severely limited by the need for low latency. Some parallelization is achieved through assigned a CPU thread to each decoder, which can then either compute inference itself or send the computation to the NPU. Another, more drastic form of parallelism is pipelining. Specifically, MTL models are ideal for splitting and scheduling, so the natural extension is to compute these pieces of architecture on different processors in parallel. Every time the encoder finishes computing a feature map, it can store it (or its address) in a shared queue that the decoders are waiting for, then request another camera frame. While the decoders are transforming the current feature map into recognizable information, the encoder is processing the next image. The amortized latency of the system is therefore $\max(E_R, D_R)$, where E_R = encoder runtime and D_R = total decoder and postprocessing runtime. Figure 4.1 shows this process are a graph between cycle numbers, where a cycle's length is equal to $\max(E_R, D_R)$, and a stream of image inputs. MTL layers that share a column are computed in parallel.

The method chosen in this work for evaluating the efficiency of the pipelining strategy is a comparison between the sequential runtime and the real frame rate in milliseconds. To provide bounds for our analysis, an ideal two stage pipeline would produce metrics twice as fast as a process with no pipeline and a pipeline is considered a net negative if it is outperformed by the baseline. This can be represented as a percentage of the maximum performance based on the runtime of the encoder, which is calculated using the equation in figure 4.2. Percentages based on both **Encoder + Decoder Time (ms)** and $2 \times$ **Encoder Time (ms)** are presented in table 4.7 since the latter metric accounts for stall time while the decoder waits for the encoder or vice versa.

Table 4.6 demonstrates why pipelining is so powerful: even with runtimes[6] greater than the runtimes in table 4.5, the frame rates of the pipelined process are consistently higher. NPU-CPU improves by 39.8%; NPU-NPU improves by 49.4%; CPU-CPU improves by 33.0%; and CPU-NPU improves by 44.5%. Compared to the baseline of

[6]Both those with just the encoder and those with the whole of inference and postprocessing.

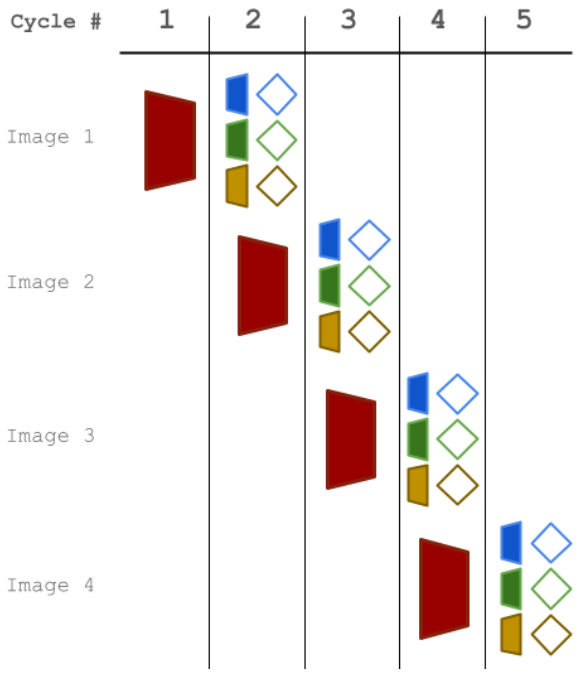


FIGURE 4.1: Two stage pipelining strategy for MTL inference.

$$P = (1 - \frac{M - \frac{1000}{\text{FPS}}}{M}) \times 100\% \tag{4.1}$$

FIGURE 4.2: Percent of ideal improvement P, where M is the maximum possible improvement.

CPU-CPU, NPU-CPU improves by 79.3%; NPU-NPU improves by 61.6%; and CPU-NPU improves by 22.8%. With these improvements applied to metrics in tables 3.4 and 3.6, a theoretical 19.04 FPS could be achieved for ResNet-18[7] and 17.11 FPS for SwinDS-Q. This is still significantly below the desired 120 FPS to prevent motion sickness in VR [42], but is a clear step closer. More experimentation will have to be done in regards to motion sickness given the vast difference in user experience between VR and AR, specifically glasses like the Vuzix M4000[8]. Both models would have latencies between 50 and 60ms.

Compared to sequential execution, energy usage and the temperature delta mostly increased for pipelined inference. For comparison, the maximum energy consumption in table 4.5 is 0.35mWh, compared to the minimum in table 4.6 of 0.36mWh. The maximum temperature delta sequentially was 0.68°C while the minimum was 0.62°C, with the next minimum at 0.74°C. Pipelining, while great for pure runtime, has a negative energy and temperature tradeoff that could lead to overheating and throttling, diminishing the potential runtime improvements.

[7] This assumes a similarity to how Transformers behave on NPUs that may not hold true for CNNs.

[8] The Apple Vision Pro, a commercial and modern AR capable device, has a base frequency of 90Hz, or 90 FPS. A table with other commercial device frame rates can be found here, though most are not AR capable: https://www.uploadvr.com/study-120fps-important-to-avoid-sickness/.

Encoder Time (ms)	**Encoder + Decoder Time (ms)**	**Frames per Second**	**Energy Consumption (mWh)**	**Temperature Delta (°C)**
Encoder on NPU, Decoders on CPU				
141.52	277.64	5.97	0.42	1.04
Encoder on NPU, Decoders on NPU				
144.62	263.40	5.38	0.36	0.62
Encoder on CPU, Decoders on CPU				
209.68	322.79	4.43	0.41	0.74
Encoder on CPU, Decoders on NPU				
222.18	326.51	4.09	0.45	0.85

TABLE 4.6: Split SwinDS-T with four possible combinations for encoder/decoder scheduling while pipelining.

Percent of Theoretical Maximum Improvement	
Encoder + Decoder Time (ms)	$2 \times$ **Encoder Time (ms)**
Encoder on NPU, Decoders on CPU	
60.33	59.18
Encoder on NPU, Decoders on NPU	
70.57	64.26
Encoder on CPU, Decoders on CPU	
69.93	53.83
Encoder on CPU, Decoders on NPU	
74.88	55.02

TABLE 4.7: Effectiveness of pipeline, according to equation 4.1.

The percent of theoretical maximum improvement P as defined in equation 4.1 tends within the 60 to 75% territory when analyzing based on ideal workload partitioning, i.e. where M = **Encoder + Decoder Time (ms)**. When calculating the true values of P, we get a range roughly between 50 and 65%. The difference between the ideal and actual values gives insight into optimal hardware scheduling for Transformer-based MTLs. NPU-CPU differs from the ideal value by only 1.15 percentage points, the smallest gap by far, and NPU-NPU differs by 6.31 percentage points. In contrast, CPU-CPU differs by 16.10 percentage points, and CPU-NPU differs by 19.86 percentage points. Smaller gaps between ideal and actual percentages suggests more balanced workloads for pipelining, which is ideal for maximum speedups. Ordering based on this difference, it would seem that running the encoder on the NPU is the most important thing to do for balanced pipelining, then running the decoders on the CPU is less important but still preferable. However, there is just not enough data to make broad generalizations for all Transformers or all MTL models, so these conclusions remain tied to specifically SwinDS-T.

Chapter 5

Conclusion

Determining how best to apply deep learning to AR-capable edge devices remains an open question. This work takes the community one step forward by providing a benchmarking framework specifically for MTL, applying scheduling and pipelining to MTL models that takes advantage of their unique structure, and recording data and analysis for some of the most common and powerful MTL models currently available. The hardware-aware improvements designed in this thesis show very promising results for Transformer architectures, even when compared to more polished CNNs that dominated vision tasks for decades. With these results and developments also come a new host of potential research directions that are worth anticipating.

5.1 Results

By combining processor-level scheduling, architecture-level pipelining, and unique benchmarking methodologies, this work was able to identify a path towards a 79.3% improvement for a Swin-based MTL model over its sequentially executed and CPU-bound baseline. This strategy got within 1.15% of the experimentally determined ideal distribution of compute in a two stage encoder-decoder pipeline. The tradeoff for these runtime improvements are energy and thermals, which is difficult to sacrifice with edge and mobile devices, especially in the face of throttling. Therefore, each reasonable scheduling distribution was thoroughly benchmarked and analyzed, allowing easy customizability for various applications. Less successful improvement methods were also discussed to help steer future development and research.

5.2 Future Work

- Software development for BeAR is essential for future research to allow for greater customizability, reliability, and usability. Details can be found on the projects GitHub page.
- The methodologies in this paper could also be applied to other models, including other Transformers or CNNs, to identify ideal high-level hardware configuration.
- Overheating and temperature throttling continue to be important, unsolved issues.

- Tying this work in with more model specific improvements, such as quantization, sparsity, and pruning could lead to even further speedups with a lower energy and thermal cost.
- Applying lower level scheduling and pipelining could lead to further improvements for specific hardware.
- Attempting these same improvements in a software other than Java could be a great novelty, especially if they still perform real-time inference with camera input.
- Minimizing latency from pipelining (even with only two stages) would lower the rate of motion sickness while using AR.

Bibliography

[1] Ahmed Agiza, Marina Neseem, and Sherief Reda. "MTLoRA: A Low-Rank Adaptation Approach for Efficient Multi-Task Learning". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: https://arxiv.org/abs/1409.0473.

[3] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[4] Han Cai et al. "Enable Deep Learning on Mobile Devices: Methods, Systems, and Applications". In: *ACM Transactions on Design Automation of Electronic Systems* 27.3 (Mar. 2022), pp. 1–50. ISSN: 1557-7309. DOI: 10.1145/3486618. URL: http://dx.doi.org/10.1145/3486618.

[5] Rich Caruana. "Multitask Learning". In: *Machine Learning* 28 (July 1997). DOI: 10.1023/A:1007379606734.

[6] Jessie Y. C. Chen and Jennifer E. Thropp. "Review of Low Frame Rate Effects on Human Performance". In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 37.6 (2007), pp. 1063–1076. DOI: 10.1109/TSMCA.2007.904779.

[7] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*. 2016, 262–263.

[8] Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. *A Survey on Deep Neural Network Pruning-Taxonomy, Comparison, Analysis, and Recommendations*. 2024. arXiv: 2308.06767 [cs.LG]. URL: https://arxiv.org/abs/2308.06767.

[9] Sofiane Chetoui et al. "ARBench: Augmented Reality Benchmark For Mobile Devices". In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2022, pp. 242–244. DOI: 10.1109/ISPASS55109.2022.00035.

[10] Hung-Lin Chi, Shih-Chung Kang, and Xiangyu Wang. "Research trends and opportunities of augmented reality applications in architecture, engineering, and construction". In: *Automation in Construction* 33 (2013). Augmented Reality in Architecture, Engineering, and Construction, pp. 116–122. ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2012.12.017. URL: https://www.sciencedirect.com/science/article/pii/S0926580513000022.

[11] Marius Cordts et al. *The Cityscapes Dataset for Semantic Urban Scene Understanding*. 2016. arXiv: 1604.01685 [cs.CV]. URL: https://arxiv.org/abs/1604.01685.

[12] Tim Dettmers et al. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. 2022. arXiv: 2208.07339 [cs.LG]. URL: https://arxiv.org/abs/2208.07339.

[13] Niloofar Didar and Marco Brocanelli. "eAR: An Edge-Assisted and Energy-Efficient Mobile Augmented Reality Framework". In: *IEEE Transactions on Mobile Computing* 22.7 (2023), pp. 3898–3909. DOI: 10.1109/TMC.2022.3144879.

[14] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *CoRR* abs/2010.11929 (2020). arXiv: 2010.11929. URL: https://arxiv.org/abs/2010.11929.

[15] Martin Eckert, Julia S Volmerg, and Christoph M Friedrich. "Augmented Reality in Medicine: Systematic and Bibliographic Review". In: *JMIR Mhealth Uhealth* 7.4 (Apr. 2019), e10967. ISSN: 2291-5222. DOI: 10.2196/10967. URL: http://www.ncbi.nlm.nih.gov/pubmed/31025950.

[16] Mark Everingham et al. "The Pascal Visual Object Classes (VOC) challenge". In: *International Journal of Computer Vision* 88 (June 2010), pp. 303–338. DOI: 10.1007/s11263-009-0275-4.

[17] Angela Fan et al. *Training with Quantization Noise for Extreme Model Compression*. 2021. arXiv: 2004.07320 [cs.LG]. URL: https://arxiv.org/abs/2004.07320.

[18] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[19] Erwan Grelier, Anthony Nouy, and Mathilde Chevreuil. *Learning with tree-based tensor formats*. 2019. arXiv: 1811.04455 [stat.ML]. URL: https://arxiv.org/abs/1811.04455.

[20] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[21] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[22] Andrey Ignatov et al. *AI Benchmark: All About Deep Learning on Smartphones in 2019*. 2019. arXiv: 1910.06663 [cs.PF]. URL: https://arxiv.org/abs/1910.06663.

[23] *Image Processing: Convolution filters and Calculation of image gradients — linkedin.com*. https://www.linkedin.com/pulse/image-processing-convolution-filters-calculation-gradients-yadav/. [Accessed 23-04-2025].

[24] Norman P. Jouppi et al. *TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings*. 2023. arXiv: 2304.01433 [cs.AR]. URL: https://arxiv.org/abs/2304.01433.

[25] Georgios Kapidis et al. *Multitask Learning to Improve Egocentric Action Recognition*. 2019. arXiv: 1909.06761 [cs.CV]. URL: https://arxiv.org/abs/1909.06761.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[27] Yann LeCun et al. "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky. Vol. 2. Morgan-Kaufmann, 1989. URL: https://proceedings.neurips.cc/paper_files/paper/1989/file/53c3bce66e43be4f209556518c2fcb54-Paper.pdf.

[28] Ze Liu et al. *Swin Transformer: Hierarchical Vision Transformer using Shifted Windows*. 2021. arXiv: 2103.14030 [cs.CV]. URL: https://arxiv.org/abs/2103.14030.

[29] Josh Millar et al. *Benchmarking Ultra-Low-Power µNPUs*. 2025. arXiv: 2503.22567 [cs.LG]. URL: https://arxiv.org/abs/2503.22567.

[30] Pushmeet Kohli Nathan Silberman Derek Hoiem and Rob Fergus. "Indoor Segmentation and Support Inference from RGBD Images". In: *ECCV*. 2012.

[31] Marina Neseem, Ahmed Agiza, and Sherief Reda. *AdaMTL: Adaptive Input-dependent Inference for Efficient Multi-Task Learning*. 2023. arXiv: 2304.08594 [cs.CV]. URL: https://arxiv.org/abs/2304.08594.

[32] Daniil Pakhomov et al. "Deep Residual Learning for Instrument Segmentation in Robotic Surgery". In: *arXiv preprint arXiv:1703.08580* (2017).

[33] Klen Čopič Pucihar and Paul Coulton. "Exploring the Evolution of Mobile Augmented Reality for Future Entertainment Systems". In: *Comput. Entertain.* 11.2 (Jan. 2015). DOI: 10.1145/2582179.2633427. URL: https://doi.org/10.1145/2582179.2633427.

[34] Jürgen Schmidhuber. "Learning to Control Fast-Weight Memories: An Alternative to Dynamic Recurrent Networks". In: *Neural Computation* 4.1 (1992), pp. 131–139. DOI: 10.1162/neco.1992.4.1.131.

[35] Zhiqiang Shen and Marios Savvides. *MEAL V2: Boosting Vanilla ResNet-50 to 80%+ Top-1 Accuracy on ImageNet without Tricks*. 2021. arXiv: 2009.08453 [cs.CV]. URL: https://arxiv.org/abs/2009.08453.

[36] Assaf Shmuel, Oren Glickman, and Teddy Lazebnik. *A Comprehensive Benchmark of Machine and Deep Learning Across Diverse Tabular Datasets*. 2024. arXiv: 2408.14817 [cs.LG]. URL: https://arxiv.org/abs/2408.14817.

[37] Shaden Smith, Jongsoo Park, and George Karypis. "Sparse Tensor Factorization on Many-Core Processors with High-Bandwidth Memory". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 1058–1067. DOI: 10.1109/IPDPS.2017.84.

[38] Tianxiang Tan and Guohong Cao. "Efficient Execution of Deep Neural Networks on Mobile Devices with NPU". In: *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (Co-Located with CPS-IoT Week 2021)*. IPSN '21. Nashville, TN, USA: Association for Computing Machinery, 2021, pp. 283–298. ISBN: 9781450380980. DOI: 10.1145/3412382.3458272. URL: https://doi.org/10.1145/3412382.3458272.

[39] Vlad Taran et al. "Performance Evaluation of Deep Learning Networks for Semantic Segmentation of Traffic Stereo-Pair Images". In: *Proceedings of the 19th International Conference on Computer Systems and Technologies*. CompSysTech'18. ACM, Sept. 2018, pp. 73–80. DOI: 10.1145/3274005.3274032. URL: http://dx.doi.org/10.1145/3274005.3274032.

[40] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[41] Oriol Vinyals et al. "Show and Tell: A Neural Image Caption Generator". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015.

[42] Jialin Wang et al. "Effect of Frame Rate on User Experience, Performance, and Simulator Sickness in Virtual Reality". In: *IEEE Transactions on Visualization and Computer Graphics* 29.5 (2023), pp. 2478–2488. DOI: 10.1109/TVCG.2023.3247057.

[43] Haocheng Xi et al. *Training Transformers with 4-bit Integers*. 2023. arXiv: 2306.11987 [cs.LG]. URL: https://arxiv.org/abs/2306.11987.

[44] Jinjie Zhang, Yixuan Zhou, and Rayan Saab. *Post-training Quantization for Neural Networks with Provable Guarantees*. 2023. arXiv: 2201.11113 [cs.LG]. URL: https://arxiv.org/abs/2201.11113.

[45] Yang Zhou et al. *Play It Cool: Dynamic Shifting Prevents Thermal Throttling*. 2022. arXiv: 2206.10849 [cs.LG]. URL: https://arxiv.org/abs/2206.10849.

Edit History

- October 8, 2025: Removed redundant table (4.6).
- October 8, 2025: Removed signature page from Tex file.