CHALMERS
UNIVERSITY OF TECHNOLOGY

Exploring Optimized CPU-Inference for Latency-Critical Machine Learning Tasks

An evaluation of CPUs as an alternative hardware for real-time computer vision applications by using model compression

Master's thesis in Complex Adaptive Systems

MAX SEDERSTEN
AMANDA SIKLUND

DEPARTMENT OF PHYSICS

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

Exploring Optimized CPU-Inference for Latency-Critical Machine Learning Tasks

An evaluation of CPUs as an alternative hardware for real-time computer vision applications by using model compression

MAX SEDERSTEN
AMANDA SIKLUND

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2024

Exploring Optimized CPU-Inference for Latency-Critical Machine Learning Tasks
An evaluation of CPUs as an alternative hardware for real-time computer vision applications by using model compression
MAX SEDERSTEN
AMANDA SIKLUND

Supervisor: Filip Wikman, Tenfifty
Examiner: Mats Granath, Department of Physics

Master's Thesis 2024
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2024

Exploring Optimized CPU-Inference for Latency-Critical Machine Learning Tasks
An evaluation of CPUs as an alternative hardware for real-time computer vision applications by using model compression
MAX SEDERSTEN, AMANDA SIKLUND
Department of Physics
Chalmers University of Technology

Abstract

In recent years, machine learning has grown to become increasingly prevalent for a wide range of applications spanning multiple industries. For some of these applications, low latency can be critical, which may limit the types of hardware that can be used. Graphical Processing Units (GPUs) have long been the go-to hardware for machine learning tasks, often outperforming alternatives like Central Processing Units (CPUs), but these are not practical in all situations. We explore CPUs, leveraging modern optimization techniques like pruning and quantization, as a competitive alternative to GPUs with comparable predictive performance. This thesis provides a comparison of the two hardware types on a real-time latency-critical vision task. On the GPU side, TensorRT in combination with quantization is used to achieve state-of-the-art inference performance on the hardware. On the CPU side, the model is optimized using SparseML to introduce unstructured sparsity and quantization. This optimized model is then used by the DeepSparse runtime engine for optimized inference. Our findings show that the CPU approach can outperform the GPU hardware in certain situations. This suggests that CPU hardware could potentially be used in applications previously limited to GPUs.

Keywords: machine learning, neural network, model compression, pruning, quantization, optimization, CPU, GPU, Neural Magic, NVIDIA

Acknowledgements

We would like to thank our supervisor at Tenfifty, Filip Wikman, for his guidance and support throughout this thesis. His expertise and insightful contributions have been a valuable part of shaping the direction and outcomes of our work.

We would also like to thank our supervisor and examiner at Chalmers, Mats Granath, for his valuable feedback and assistance, particularly in providing insightful guidance that helped us shape the project outline.

Max Sedersten and Amanda Siklund, Gothenburg, June 2024

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
AP	Average Precision
CNN	Convolutional Neural Network
CPU	Central Processing Unit
GPU	Graphical Processing Unit
IoU	Intersection over Union
mAP	mean Average Precision
OBD	Optimal Brain Damage
OBS	Optimal Brain Surgeon
OKS	Object Keypoint Similarity
ONNX	Open Neural Network Exchange
PTQ	Post Training Quantization
QAT	Quantization Aware Training
ReLU	Rectified Linear Unit
TPU	Tensor Processing Unit
YOLO	You Only Look Once

Contents

List of Figures

List of Tables

1

Introduction

Today, machine learning is a rapidly evolving field with uses in a wide range of industries. The recent introduction of ChatGPT by OpenAI last year has catapulted the term "AI" into the spotlight, resulting in a lot of buzz around the topic [1]. Companies have also started to notice, with many committing large amounts of capital to their own AI research, so much so that the main supplier of AI hardware, NVIDIA, has hit an evaluation of over 2 trillion dollars [2].

NVIDIA's position in the market is no coincidence. The company's primary focus has for a long time been Graphical Processing Units (also known as GPUs), which is hardware specifically designed for graphically intensive workloads. These types of computational workloads are often heavily parallelizable in nature, a fact that is also true for many machine learning workloads. Common neural network structures such as Convolutional Neural Networks (CNN), and standard fully connected networks, are implemented based on matrix operations that can take advantage of the parallelization capabilities of a GPU. GPUs, with their relatively large and high-speed memory, offer the advantage of running and training models without constantly reading and writing from slower memory. As a result, GPUs often outperform other types of hardware such as Central Processing Units (CPUs), both in terms of latency and throughput, for most machine learning applications.

But what is optimal also depends a lot on the application in question and the surrounding limitations. In some situations, power consumption might be the biggest concern, in others it might be the cost. Yet others put demands on latency which might completely change the appropriate hardware. For example, real-time applications might be limited to running on edge devices since the server latency might be too high [3]. This in turn may come with power limitations while at the same time being limited to the hardware that is already available.

Yet another factor that could change the appropriate hardware type is model optimization techniques. Model optimization, in the context of machine learning, is the process of making changes to a model (or runtime) so that it runs faster and/or more efficiently on a given set of hardware, without significantly altering the model's behavior. There are several different approaches to model optimization, with a promising sub-field being model compression. The use of compression

techniques – like pruning and quantization – can yield considerable speed-ups for inference on some hardware types while also reducing the storage footprint.

1.1 Background

For decades, researchers have worked to emulate the brain's complex functions using artificial systems. These efforts inspired to the creation of Artificial Neural Networks, which are network structures with nodes representing neurons and weights representing connections. These structures are a cornerstone of modern machine learning and have proven to be highly capable. However, over time, a trend of larger and larger network sizes has emerged resulting in higher computational demands.

In response, the biomimicry-based concept known as pruning, which draws inspiration from the brain's ability to refine its neural connections over time, has been explored as a way of making models more efficient [4]. In the context of neural networks, pruning is the process of removing specific weights and/or neurons based on their importance, in order to simplify the model.

Previous studies have proposed different pruning techniques with very promising results, as shown in [5]. It demonstrates that pruning unnecessary weights can give great results, with large sparse models outperforming small dense ones of the same size, on multiple fronts. Other previous works have explored alternative approaches for determining what to prune in the network, such as removing individual nodes or groups of nodes [6], [7], [8]. These still face a few challenges, however. Aggressive pruning may incur significant information loss which in turn can impact the predictive performance.

The theoretical speed-ups possible through pruning have typically only been realized using coarser pruning strategies or implementations leveraging very specific network structures, but this is not true for the case of SparseML. SparseML – developed by Neural Magic – is an optimization library that compresses models for efficient inference on CPU using their runtime engine: DeepSparse [9]. It achieves substantial speed-ups on a wide range of model types with the use of fine-grained pruning strategies that can be efficiently utilized by the underlying structure of a CPU, something that can not be said about GPUs.

CPUs – like the ones powering computers and smartphones – are more versatile in nature and are thus able to handle a wider variety of computations. They usually have a smaller number of more powerful cores compared to the relatively high number of "simpler" cores found in GPUs. The reason for this is that CPUs are optimized for sequential workloads, unlike the parallelization approach of GPUs. This – in combination with the fact that CPUs' dedicated memory is relatively small – results in a lot of inefficient memory transfers, often with the same data being shuffled in and out of memory multiple times during inference. This makes CPUs inferior for most machine learning applications, with their only advantage being their versatility and widespread use. However, Neural Magic claims they are able to

provide GPU-level inference speed on CPUs thanks to their optimization techniques [10].

Another notable example of machine learning hardware is Tensor Processing Units (TPUs), developed by Google. TPUs are Application-Specific Integrated Circuits (ASICs) specifically designed for use with TensorFlow [11]. These excel at doing large matrix operations at scale and can outperform CPUs and GPUs in regards to throughput on some specific workloads [12]. However, this speed comes at a cost. Factors such as model size and data constraints might have to be considered since TPUs only realize their speed-ups for bigger models and at larger batch sizes, making them impractical for some applications. The model architecture itself can also have a major impact on inference performance with some architectures not being supported at all. This limits the situations where TPUs can effectively be utilized and adds to the complexity of choosing hardware.

The GPU has also seen advancements in recent years. Particularly with NVIDIA's introduction of Tensor Cores [13] together with their inference library: TensorRT [14]. Tensor Cores are specially designed processing units that are well suited for computing parts of neural networks that depend heavily on matrix operations. TensorRT provides the tools needed to convert a model into a format compatible with Tensor Cores, while also providing a compatible runtime for execution.

1.2 Gap

Both Neural Magic and NVIDIA make claims about outperforming the other with their proprietary technologies [15] [16]. But despite their claims, there exists no comprehensive comparison that accounts for both optimization techniques as well as recent technological advancements. A reason for this could be that each party tries to showcase its strengths while not shining any light on its weaknesses. NVIDIA for example, when comparing their machine learning accelerators to CPUs, often uses examples that are well suited for parallelization in combination with larger batch sizes when computationally applicable [17]. This benefits NVIDIA since GPUs generally scale well with larger batch sizes, resulting in relatively little additional overhead as long as hardware limits are not exceeded.

This showcased inference performance is not in any way incorrect, but it can still be misleading, particularly for the use-case studied in this thesis which is real-time applications. For real-time latency-critical computer vision applications, batching is unlikely to decrease the latency, at least as long as computational power is not the main bottleneck. The last frame in a given batch would, in the ideal case, take at least the same time as computing all the frames sequentially. The first frame meanwhile would have to wait for all subsequent frames in the batch to be created, before processing can start, adding substantially to that frame's absolute latency. This means that the results presented are not a fair indication of the real-world performance for real-time applications like the one explored in this thesis.

Neural Magic on the other hand shows results relevant to the task but it is still lacking in some areas [18]. Firstly, they fail to take recent technological advancements by NVIDIA into account in their comparison. Their optimization techniques may also have an impact on the model's predictive performance, which is not well represented in their results.

In conclusion, there seem to be no publicly available apples-to-apples comparisons of current state-of-the-art GPU technology versus traditional CPU hardware empowered by modern optimization techniques.

1.3 Aim

This thesis aims to determine if and when CPUs can be a competitive alternative to modern GPUs, for a given real-time computer vision task, with the aid of optimization techniques. The aim is also to objectively showcase the trade-offs between inference speed and predictive performance inherent with these optimization techniques.

1.4 Limitations

This work focuses on the optimization of a convolution-based pose prediction model for use with real-time applications, without considering other types of models, tasks, or network architectures. Additionally, the scope of this work is limited to the model itself, without considering the performance impacts of other parts of the pipeline such as pre-processing and post-processing.

The chosen model is also the base for all evaluated model versions with the unoptimized performance serving as the baseline that the results are compared against. No comparison with external benchmarks is performed.

The hardware used for evaluation is also limited due to availability. On the GPU side, a Nvidia Jetson AGX Orin is the only hardware used, while on the CPU side, Google Cloud instances limited to a single line of processors are evaluated. Cloud instances with varying numbers of cores are however considered and evaluated.

The optimization techniques explored are limited to pruning and quantization, with pruning itself being limited to unstructured pruning. Structured pruning is discussed but not evaluated since improvements would apply to both hardware. Neither is semi-structured pruning explored due to the minimal impact on inference speed when small batch sizes are used, as is common for real-time applications [19].

1.5 Research Questions

This thesis aims to answer the following questions:

- Can CPUs be a competitive alternative to GPUs for latency-critical applications when using compression techniques?

- What is the predictive performance impact resulting from the use of compression techniques?

2

Theory

This section covers the theoretical background relevant for the work being done in this thesis. The areas covered mainly revolve around model compression as well as topic relating to model inference.

2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a special type of feed-forward neural network often used for vision tasks like image classification and object detection [20]. These networks have a fewer number of connections (also known as weights) than many other types of neural networks, like fully connected networks, with the same number of neurons. This is due to how the layers within CNNs, called convolutional layers, work. These layers leverage kernels, or filters, which are groups of learnable parameters that slide along the spatial dimensions of the input. In the context of vision-based CNNs, this often involves the use of 2D convolutions where the input consists of two spatial dimensions together with a third optional dimension that represents the color channels, as visualized in Figure 2.1. As the kernel slides along the spatial dimension, it convolves with that part of the input to produce an output corresponding to that region. This essentially means that these weights are being shared among multiple output neurons, leading to the reduction in parameter count.

In modern CNN architectures for tasks such as image classification, object detection, and pose estimation, the network is often divided into two main components: the backbone and the head. The backbone serves as the foundational component of the network, responsible for feature extraction from the input data. It consists of a series of hierarchically arranged convolutional layers, designed to capture features from the raw input image or data. The head component is responsible for task-specific processing, taking the features extracted by the backbone and transforming them into predictions relevant to the task at hand. For instance, in object detection, the head may include layers for bounding box regression and classification, while in pose estimation, it may involve layers for keypoint detection and association.

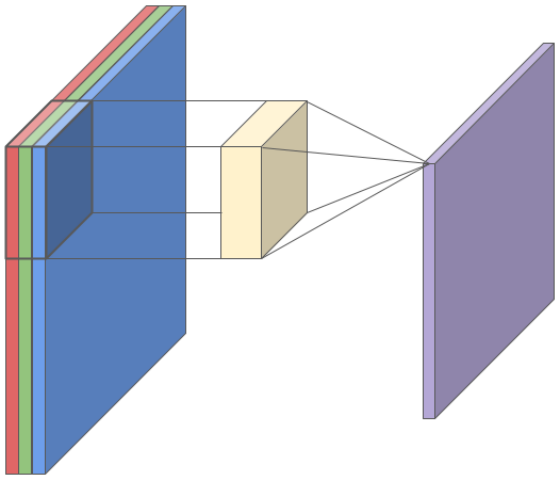

Figure 2.1: Visualization of a 2D convolution, including the input matrix (left), the convolutional kernel (center), and the resulting output matrix (right).

2.2 Pruning

Pruning is an optimization method for neural networks first introduced in [4], but has since then evolved considerably. The general idea behind pruning is to remove nodes and/or weights that are deemed to have low importance for a model's output. There are a few different approaches that work in slightly different ways and with slightly different goals in mind, but the process can generally be split into three overarching steps: weight/node selection, the actual pruning, and fine-tuning.

When it comes to the selection of weights/nodes, there are also a few different methods that work in slightly different ways. Optimal Brain Damage (OBD), introduced by the paper of the same name [4], is a selection method based on the loss function of the model. This method aims to find the weights that have the lowest impact on the loss once removed. Computing the loss impact of every node is not always feasible, however. To solve this, OBD uses a local approximation of the loss function to determine the weights that should be selected.

Magnitude-based methods on the other hand are a lot simpler in nature. These methods build on the assumption that weights that shrink during training and end up small, likely have low importance for the final output. Although this is not always the case, it is a good enough approximation that works fairly well, especially compared to purely random methods relying on pure chance alone [21]. Although magnitude-based methods are relatively simple, they have the added benefit of being easier to compute in most situations, while not lagging too far behind other methods [22].

Once the nodes/weights to prune have been established, the pruning can take place

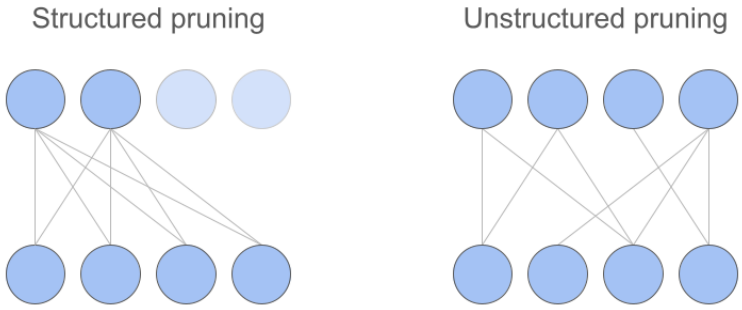

Figure 2.2: Illustration of structured and unstructured pruning. Left side shows the removal of structured groups of weights, while the right side shows the removal of weights in a unstructured manner.

and this, in turn, can be done in a few different ways. The different pruning approaches differ fundamentally from each other, with their advantages, disadvantages, and use cases. The different types of pruning approaches are described in more detail in the following sections.

2.2.1 Unstructured pruning

Unstructured pruning is an inherently fine-grained approach with its focus on individual weights, as visualized in Figure 2.2. It revolves around setting individual weights to zero (0), thus effectively removing them in place without making changes to the underlying structure of the network. The resulting model is called a "sparse model" since it is no longer densely connected (although it may still technically be due to weights only being masked).

Sparse models have some big implications when it comes to model inference. Any contribution to the output from nodes involving a masked weight is always known to be zero and can thus be ignored during computation. This could in theory provide a substantial boost in inference performance, especially at the sparsification levels shown to be practical in previous work [4] [23]. In practice, however, it is not that simple.

Hardware type plays a major role in the effectiveness of fine-grained pruning strategies. GPUs, as mentioned in Section 1, rely heavily on parallelization due to how they are fundamentally designed to operate [24]. This design provides great performance for inference and is one of the reasons they are so dominant for machine learning tasks. But one thing they tend to struggle with is the utilization of sparse computations. The computation time for a parallel computation is nearly constant, as the number of operations done in parallel scale. This means that on hardware that leans heavily on parallelization, like GPUs, pruning provides little to no benefit in terms of speed-ups.

Pruning also has other benefits. It allows for a smaller storage footprint than their

dense counterpart [5]. But this size reduction depends heavily on how the sparse model is stored as well as its sparsification level. Since the sparsification itself also adds some storage overhead, the reduction may vary.

2.2.2 Structured pruning

Structured pruning is a coarser approach where collections of weights that make up structural parts of the neural network are removed [25], as shown in Figure 2.2. There are a few different types of structured pruning that trim different parts of the structure and are suited for different types of model architectures. A simple approach specifically relevant for CNNs is filter pruning [26].

Filter pruning, as the name suggests, involves the removal of entire filters, a process that does not introduce any sparsity and leaves the model dense. This means the model can be computed in the same way as before, which has the benefit of not requiring any specialized hardware or software implementations to be utilized. Any theoretical speed-ups will be realized on any previously compatible hardware, allowing for painless deployment.

There are, however, some major drawbacks. Structured pruning is inherently a lot coarser in nature than unstructured pruning [24]. The least important collection of weights may still contain important information in some weights that will be discarded along with the rest of them. This results in quicker drops in predictive performance as compression levels increase, compared to unstructured pruning, which limits the amount of pruning that can practically be applied through structured pruning.

2.2.3 Pruning-related fine-tuning

Pruning is most often accompanied by training or fine-tuning. When pruning a model that will be trained from scratch, the pruning is typically incorporated into the main training loop so that it is performed iteratively throughout the training process [4]. Similarly, when pruning a pre-trained model, it is often done in an iterative process, alternating between pruning and fine-tuning. The reason for this is that the removal of weights can, and likely will make the model diverge from the previous solution. Thus there is the need for re-calibrations to compensate for this change, a process that requires a dataset.

The hope is that the model retains the intended information while discarding redundant information in the process. This is the reason why it is often done iteratively and not all at once. When pruning iteratively, the information can be distilled between pruning steps [27]. This in turn changes the state of the model which can change the weight magnitude distribution completely. For example, weights that were initially small but not small enough to be pruned can come to contain distilled information after the fine-tuning, either directly from the information contained in the removed weights or simply due to the changing needs of the network around it.

2.2.4 Post-training pruning

Although pruning often is done iteratively, in tandem with training, it can still be done in a single step. This is referred to as one-shot or post-training pruning, and this approach has the added benefit of being a lot simpler to implement and compute. Where it falls short however is predictive performance [28]. One-shot pruning tends to incur higher losses in predictive performance compared to gradual pruning techniques at the same compression rate.

2.3 Quantization

Neural networks can be quantized to reduce the numeric precision of the network's weights and activations. This process involves representing the weights and activations with lower bit-widths (e.g., 8-bit integers instead of 32-bit floating-point numbers), with the primary goal of reducing the computational and memory requirements of neural networks. It can significantly enhance the inference performance of neural networks, leading to faster inference times, lower power consumption, and reduced memory usage. This has been shown to greatly increase computational efficiency on a wide variety of models to a high degree, with minimal impact on precision [29]. There are two main ways that quantization can be applied, namely, post-training quantization (PTQ) and quantization-aware training (QAT).

The first step of quantization is typically to determine the fixed parameters such that it minimizes the information loss during conversion. These parameters vary based on the conversion technique used, with the main two being symmetric and asymmetric quantization. Symmetric quantization uses the absolute maximum value to map the weights/activations, while asymmetric quantization additionally uses the zero-point value.

Absolute maximum quantization minimizes the information loss by mapping the absolute maximum value of all weights to the min/max values of the quantized range. This is done by dividing the maximum original value, $|r_{\max}|$, and scaled by a factor S. This process can be represented as

$$q = S\frac{r}{|r_{max}|} \tag{2.1}$$

Here, q represents the quantized value, r the original value, and S the scaling factor.

Zero-point quantization is typically used when dealing with an asymmetric distribution. This could for example occur when dealing with only positive values, such as those resulting from the ReLU (Rectified Linear Unit) function. ReLU is an activation function replacing all negative input values with zero and is very common for CNNs.

The conversion in zero-point quantization includes two types of parameters: one for scaling the values and another representing the zero-point value. A representation

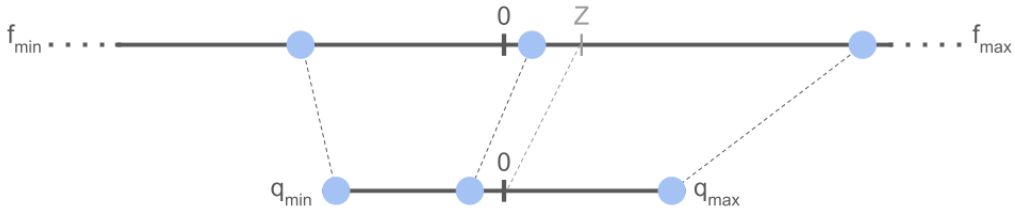

Figure 2.3: Visualization of zero-point quantization, illustrating the mapping of a floating-point distribution onto a lower-precision representation. Z represents the zero-point value.

of how the values are scaled and mapped in relation to the zero-point value is visualized in Figure 2.3. This example demonstrates a skewed floating-point distribution mapped onto a representation with lower resolution, which includes a zero-point value Z. The relationship between the original values r and quantized values q is calculated as

$$r = S(q - Z) \tag{2.2}$$

which includes both the scaling factor S and the zero-point parameter Z.

2.3.1 Post Training Quantization (PTQ)

Applying quantization after training is a simpler approach compared to QAT because it does not require adjusting the training process to account for quantization [30]. Instead, quantization is applied to the model's weights after training. However, there are potential drawbacks to this approach. Since quantization is not considered during training, applying it to an already trained model may lead to severe precision loss. To address this quantization error, the quantized model can be fine-tuned to compensate for this loss in accuracy. This will however not yield the same result as through the use of QAT.

2.3.2 Quantization Aware Training (QAT)

A challenge encountered during quantization is the drop in model accuracy resulting from the reduced numeric precision of weights. This is addressed by emulating the impact of the lower precision to make the model account for these changes during training. This is done by introducing augmenting operations that emulate the effects of quantization during the forward pass, without modifying the numeric precision of the parameters or the rest of the training process [30].

An example of quantizing a convolutional layer is visualized in Figure 2.4. Typically, the weights are quantized before being multiplied or convolved with the input, while the outputs are generally quantized after the activation function. This approach is beneficial since the activation function is fused with the main operation in the most optimized hardware setups. The same approach is applied to other layers, such as concatenation and addition, allowing the model to adjust the parameters for lower precision.

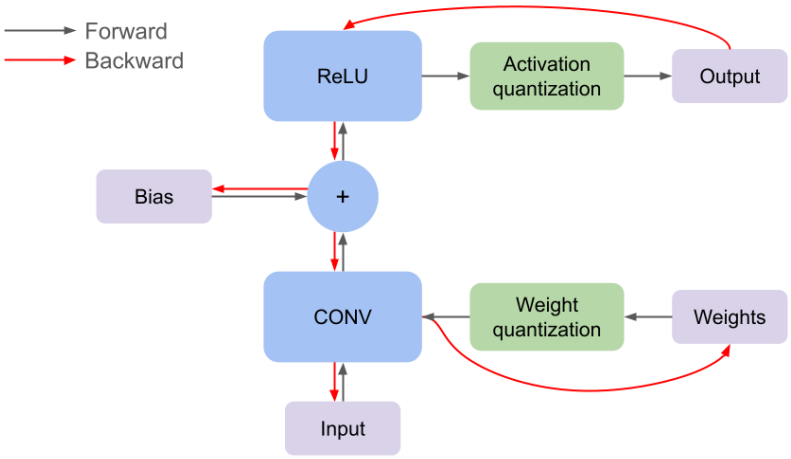

Figure 2.4: Illustrated example of a convolutional layer with QAT integration.

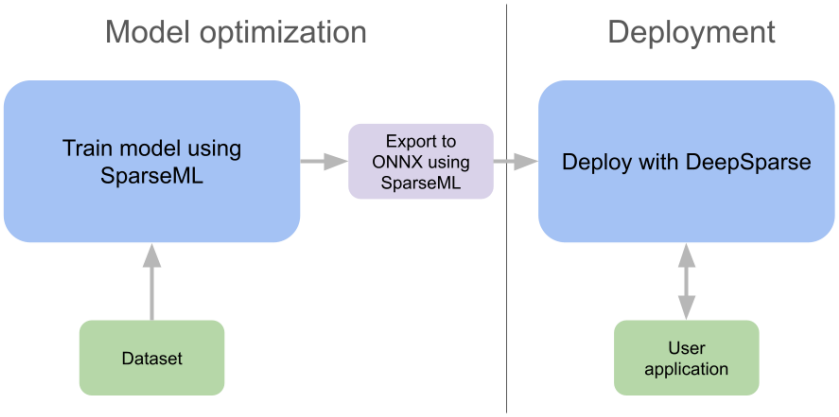

Figure 2.5: Flowchart of the model optimization process when using SparseML and DeepSparse.

When propagating backward, the gradients of the loss with respect to the model parameters are computed while skipping the quantization step. This allows the model to optimize its parameters while considering the effects of quantization.

2.4 SparseML

SparseML is an optimization library developed by Neural Magic that attempts to leverage unstructured pruning together with other compression techniques for high inference performance on CPUs [10]. Models that are compressed using SparseML are specifically optimized for use with using their runtime engine DeepSparse [18], designed to take advantage of the underlying structure of CPU hardware. A diagram for the sparsification process is shown in figure 2.5.

There are two different approaches to how SparseML can be used. One involves fine-tuning an already pruned model, while the other involves sparsifying a model from scratch. The already sparsified models, along with their corresponding sparsification recipes, are available at Neural Magic's repository called SparseZoo [31]. These models can be retrained on a new task or utilized as they are.

SparseML integrates with various frameworks, such as PyTorch and TensorFlow, by utilizing the callbacks integrated into their training processes to apply the compression. Since most frameworks already have these callbacks, no modifications to the framework are needed and the training process can proceed as normal with compression.

The compression applied during training is configurable and can be defined in a file known as a sparsification recipe. Besides specifying training-related parameters such as learning rate and number of epochs, it also specifies if and how pruning and optimization should be applied. The types of compression that SparseML supports include both gradual and one-shot pruning techniques. Additionally, it supports both QAT and PTQ, with QAT being the dominant method used in Neural Magic's model repository: SparseZoo [31].

Typically, when applying both pruning and quantization, the process involves initially stabilizing the model for a few epochs, followed by a gradual application of pruning, and then lastly applying quantization, accompanied by calibration for the case of QAT.

2.5 DeepSparse

DeepSparse is a runtime engine developed by Neural Magic that is optimized for running sparse models on x86 CPUs. It achieves speed-ups through something called Column Tensors [18]. Column Tensors are isolated groups of connected neurons within the neural network that span it in a depth-wise fashion as visualized in Figure 2.6. Unstructured pruning introduces "cavities" into the network structure, allowing it to be broken into column tensors, something that can not be done with an unpruned model due to its densely connected nature. For dense networks, no simpler representation exists, so no simplifications can be made.

Column tensors are well suited for the x86 architecture since they are often small enough to fit entirely into the CPU cache, a limiting factor of CPUs as previously mentioned in Section 1.1. The isolated nature of these Column Tensors also enables them to be computed completely independently on a single processor core. Therefore, these computations can be effectively distributed among the small number of available cores typical for CPUs, thus speeding up computation.

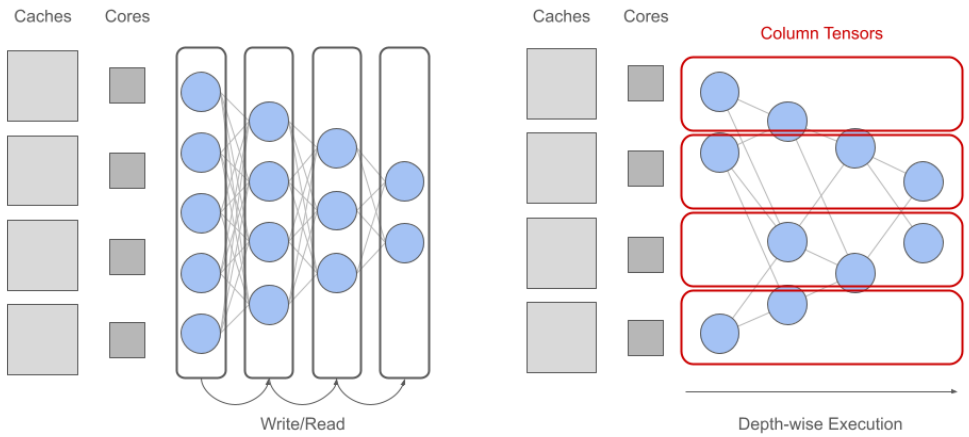

Figure 2.6: Visualization of how DeepSparse utilizes sparsity on CPU hardware. On the left, traditional execution on CPU is shown, while on the right, the sparse model is decomposed into column tensors that can run independently on individual cores.

2.6 TensorRT

TensorRT is a software development kit developed by NVIDIA for optimizing deep learning models for deployment on NVIDIA GPUs. The optimization done by TensorRT takes advantage of a special type of processing unit called Tensor Cores and was first included with NVIDIA's Volta architecture in 2017 [32].

Tensor Cores are specially designed hardware that performs fused multiply-add computations in a single step. These fused multiply-add operations can be utilized to quickly solve matrix operations, like the ones used for efficient computation of some neural network structures like convolutional layers [33]. The fused multiply-add operation takes three 4x4 matrices, denoted as $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$, and the output $\boldsymbol{D}$ calculated as

$$\boldsymbol{A}_{4\times4} \cdot \boldsymbol{B}_{4\times4} + \boldsymbol{C}_{4\times4} = \boldsymbol{D}_{4\times4} \tag{2.3}$$

Although the unit only performs these operations on matrices of size 4x4, it is still able to accelerate some workloads considerably. NVIDIA claims up to six times the number of floating-point operations per second (FLOPS) for inference compared to the previous generations of hardware [32] and research has also shown inference to be around 65% [34] faster on the same hardware when Tensor Cores are utilized compared to without. Convolution-based neural networks are also well suited to take advantage of this type of hardware since the computation of their convolutional layers effectively can be represented by matrix multiplications [33].

The optimization tools provided by TensorRT help facilitate the conversion process that is needed to make the model suitable for execution with the provided TensorRT runtime. This is done by fusing parts of the neural network structure (such as convolutional layers) into their Tensor Core compatible counterparts.

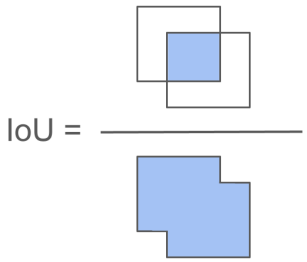

Figure 2.7: Visualization of IoU, being the ratio between the intersection area and the union area between two bounding box instances.

2.7 ONNX Runtime

Open Neural Network Exchange (ONNX) is an open-source machine learning format that is supported by a wide range of frameworks [35]. The idea behind ONNX is to allow for greater interoperability between different ecosystems encouraging collaboration.

ONNX Runtime is a complement to ONNX, functioning as an inference engine used to deploy deep learning models represented in this format. This runtime supports several different hardware platforms including CPUs, GPUs, and other specialized accelerators. The functionality revolves around optimizing the neural network graph by partitioning it into subgraphs such that it fits the specialized hardware.

2.8 Pose Metrics

Performance metrics are used to measure the predictive performance of neural networks using a number of indicators. In pose estimation, which involves the identification of keypoints often corresponding to specific body parts, determining what constitutes a correct prediction can be challenging. For instance, an eye is a distinct feature and is easier to predict with high certainty compared to the ambiguity associated with the actual location of a shoulder. The correctness of the eye's prediction is therefore more important than that of the shoulder. A commonly used metric that accounts for this difference is Object Keypoint Similarity (OKS). It considers the distance between the ground truth keypoint and the predicted keypoint, weighted by both the keypoint type as well as accounting for the scale of the object being detected [36].

The process of calculating OKS involves taking the Euclidean distances d_i between each ground truth and predicted keypoint, then determining the similarity score KS_i for each of them. The KS_i is calculated by taking the probability density of a Gaussian distribution with standard deviation sk_i evaluated at d_i. The standard deviation sk_i consists of two variables: s representing the detected object's relative size in the input and k_i defining the per-keypoint constant which describes the point's

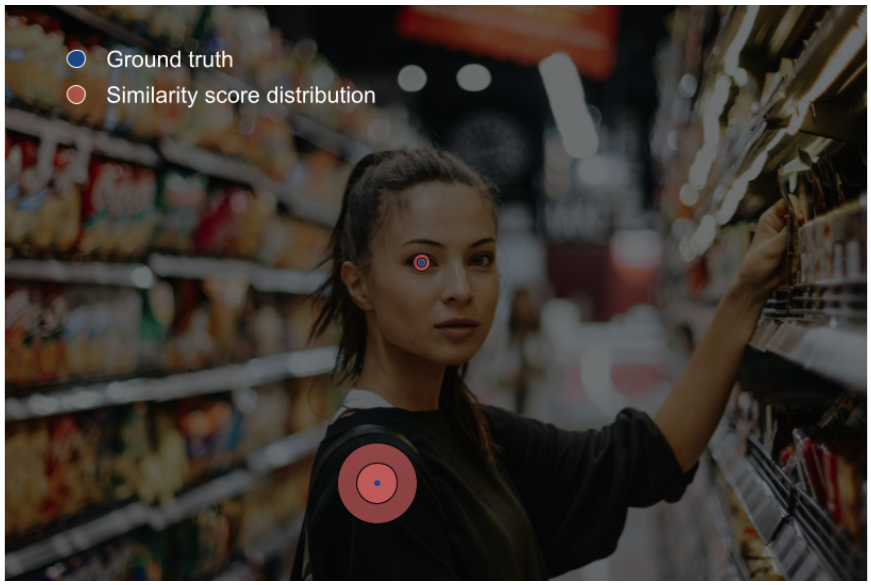

Figure 2.8: Visualization of the similarity score distribution of two different point types. Similarity score above 0.5 is represented by the red inner ring. Original image by Franki Chamaki on Unsplash [37].

ambiguity. Figure 2.8 visualizes the tolerance threshold of different keypoint types. The equation for calculating KS_i is

$$\mathrm{KS}_i = \exp\left(\frac{-d_i^2}{2s^2k_i^2}\right) \tag{2.4}$$

This approach provides a representation of similarity, with higher probabilities indicating closer agreement between the predicted and ground truth keypoints. Typically, k_i is tuned by measuring the per-keypoint standard deviation with respect to the object scale across a certain number of images from the dataset. This accounts for the scale of the keypoints in the performance measurement.

The total OKS is calculated by taking the arithmetic average of these KS_i values using Equation 2.5. Here, v_i represents the ground truth visibility flag, resulting in v_i being 1 if the keypoint is labeled and 0 otherwise. If a keypoint is not labeled ($v_i = 0$), it does not affect OKS, resulting in the average being taken only of ($v_i = 1$). The equation for OKS is defined as

$$\mathrm{OKS} = \frac{\sum_i \mathrm{KS}_i v_i}{\sum_i v_i} \tag{2.5}$$

In addition to keypoint predictions, some pose models also offer bounding box predictions. The evaluation of these bounding box predictions often involves the intersection over union (IoU) metric [36]. IoU involves defining the ratio between the

intersection area and the union area between two bounding box instances, as shown in Figure 2.7. This metric ranges from 0 to 1, where 1 signifies perfect overlap and 0 indicates no overlap.

The IoU and OKS metrics are used to measure the "correctness" of the box and pose predictions respectively. A value above a certain threshold, typically 0.5, signifies a correct prediction, also referred to as a true positive prediction. In this context, precision measures the ratio of true positive predictions to the total number of predictions (true positives+false positives), indicating the proportion of correct predictions. Mathematically, it is defined as

$$\text{Precision} = \frac{\text{True positive}}{\text{True positive} + \text{False positive}} \tag{2.6}$$

Another commonly used metric is recall, which signifies the model's capability to detect all relevant labels. It calculates the ratio of correctly predicted instances (true positives) to the total number of ground truth instances (true positives+false negatives). This metric is given as

$$\text{Recall} = \frac{\text{True positive}}{\text{True positive} + \text{False negative}} \tag{2.7}$$

Achieving both a high recall and precision is desirable as it indicates that the model's predictions are correct and that most labels are detected. However, there is an inherent tradeoff between precision and recall depending on the chosen confidence threshold. To evaluate the balance between them, a precision-recall graph can be plotted across various confidence thresholds at a specific value of IoU/OKS. The area under this curve, referred to as the Average Precision (AP) at that particular value of IoU/OKS, is defined as

$$\text{AP} = \int_0^1 p(r)dr \tag{2.8}$$

For models with multiple classes, the mean Average Precision (mAP) is used to provide a comprehensive measure of AP. mAP is calculated by averaging the AP across all classes (N), defined as

$$\text{mAP} = \frac{1}{N}\sum_{i=1}^{N} \text{AP}_i \tag{2.9}$$

This results in a single metric that reflects the model's overall precision.

Figure 2.9: Example of the output predicted by the YOLOv8s pose model, including detected keypoints, bounding boxes, and the associated confidence scores. Original image by John Doe on Unsplash [40].

2.9 YOLOv8 Pose

The YOLOv8 pose model [38] developed by Ultralytics is a variant of the YOLO (You Only Look Once) algorithm [39], which is an object detection algorithm that predicts bounding boxes from an image. YOLOv8 pose is a pose estimation model that comes in a few different sizes, with the small variant having 11.2 million parameters. These models are trained using the COCO pose dataset [36], which is comprised of 200,000 labeled images of human poses. The model's objective is to predict the poses of all humans present within an image. This is done by predicting the location of a set of keypoints that correspond to specific body parts, as visualized in Figure 2.9. Each detection includes a bounding box along with 17 keypoints, accompanied by its corresponding confidence score. The pose model is structurally similar to the YOLOv8 detect models, with the addition of a few layers at the end of the network that handle the prediction of keypoints, as well as a slight difference in parameter count in some layers.

The loss function for the YOLOv8 pose model involves a combination of components designed to optimize both bounding box detection and keypoint estimation. There are two types of loss for both boxes and poses: localization and confidence. Localization loss measures how accurately the model predicts the coordinates while confidence loss penalizes incorrect predictions. This confidence loss is computed using binary cross-entropy, where the model is trained to predict whether an object is present in each grid cell and how confident it is in that prediction. Binary

cross-entropy involves quantifying the difference between predicted and labeled probabilities for each class to penalize the model more heavily for incorrect predictions [38].

The metrics measured during model validation include precision, recall, and mean Average Precision (mAP). These metrics are computed individually for both bounding box and keypoint predictions. In Ultralytics' validation function, it typically includes two different mAP metrics: mAP50 and mAP50-95. mAP50 is a measure of the mean average precision of predictions over a threshold of 0.5, while mAP50-95 measures the mean average precision of the model's predictions across a range of IoU and OKS thresholds, from 0.5 to 0.95. This broader assessment evaluates how well the model detects objects or keypoints across different levels of overlap with ground truth annotations.

3

Methods

To evaluate the capability of CPU being a competitive alternative to GPU for real-time applications, a few different approaches and runtimes were tested. These include DeepSparse, designed for accelerating computations on CPU, TensorRT, a leading-edge technique for fast GPU inference, and ONNX Runtime representing the baseline GPU performance.

The base model used for these experiments was the YOLOv8s-pose model developed by Ultralytics. This is a pose estimation model predicting human keypoints, as described in section 2.9. This model was chosen for its efficiency and balance of speed and accuracy, making it well-suited for real-time computer vision tasks in resource-constrained environments.

3.1 CPU evaluation

SparseML [10] and DeepSparse [18] were used for speeding up the inference on CPU due to their state-of-the-art capabilities within this field. This process involved retraining the YOLOv8 pose model with optimization techniques, followed by validation on specific hardware to evaluate inference performance.

3.1.1 Hardware

For the CPU-based optimization to reach its full potential, the hardware had to be taken into consideration. Quantization performed using SparseML is intended for use with DeepSparse which relies on specific x86 instruction set extensions to work. The ones supported by DeepSparse are AVX2, AVX512, and AVX512 VNNI, with particular optimization for AVX512 VNNI specifically [41]. Although AVX2 and AVX512 are supported through emulation, their performance may not be as efficient.

For the experiments, we selected Google Cloud instances equipped with AMD Genoa hardware, which supports AVX512 VNNI. This had the added benefit of enabling an evaluation of different configurations by just changing the number of cores while keeping other factors fixed. The CPU configurations tested included setups with 8,

15, and 30 cores, as these were the available options for this instance.

3.1.2 Implementation

For pruning and quantization to be applied using SparseML, the model in question has to be supported. SparseML has built-in support for YOLOv8 models, but this implementation was based on an Ultralytics version from before the introduction of pose detection, thus it lacks support for the pose task. This lack of support also meant that there were no pre-sparsed pose models available on Neural Magic's model repository: SparseZoo. Therefore, the model had to be trained/pruned using SparseML from scratch.

Furthermore, newer versions of SparseML (version 1.6.4 and later) also had a previously unreported bug that broke quantization for exported models. This bug was confirmed by the support from Neural Magic, and therefore, a modified version based on an older version had to be created specifically for exporting the trained models.

We settled on implementing two different versions of SparseML due to the limitations mentioned above, one for pruning/training of the model, and the other for exporting it. This involves the following implementations in more detail:

- **Pruning/training modifications**
  To sparsify and/or quantify the YOLOv8 pose model, a newer version of SparseML (1.7.0) was utilized. This version depended on a version of Ultralytics (8.0.124) that contained the code related to the pose task. Therefore, an augmented version of SparseML was created by manually overwriting the validation and training classes. These custom class implementations re-used code for the pose functionality already present in Ultralytics.

- **Export modifications**
  When it came to exporting the models, an older version of SparseML (1.5.4) had to be used, with the reason being a bug that broke quantization in the newer versions. This version of SparseML had a dependency on a version of Ultralytics (8.0.30) from before the introduction of the YOLOv8 pose model, which had a completely different file structure, complicating the implementation process and making it different from the other implementations. Starting from these versions of SparseML and Ultralytics, a modified version of each was created exclusively to facilitate the export process of the models. The modifications entailed the re-implementation of some pose-related functionality as well as extending the existing code to handle pose models.

Using a newer version of SparseML only for training, with its dependency on an Ultralytics version supporting pose, was deemed to be of enough benefit to be justified. It made the implementation process a lot simpler by enabling most functions and surrounding code to be reused in the new implementation. In the case of the

version intended for exporting the model, the work was also deemed to be simpler compared to doing a combined implementation with both training and exporting. Only code relating to the export process had to be modified, saving a lot of time and effort.

The sparsed and quantized models were then validated using DeepSparse. Since DeepSparse, similar to SparseML, has built-in support for YOLOv8 but does not inherently support the pose task, this had to be implemented. This involved the following adjustments in more detail:

- **Validation modifications**
  The pose functionality had to be implemented into DeepSparse (version 1.7.1) to enable validation of the sparsed and/or quantized pose model. This version of DeepSparse depends on an Ultralytics version (8.0.124), which includes the pose estimation task. Therefore, an augmented version of DeepSparse was created, similar to the one made with SparseML. Specifically, this involved overwriting the validation class with a modified version, leveraging the existing pose-specific functionality in Ultralytics.

3.1.3 Sparsification recipes selection

Since no pre-sparsed pose models existed, the model had to be optimized from scratch through the use of sparsification recipes in SparseML. The choice of recipes was in turn influenced by several factors, with one being the existence of recipes for related models. SparseZoo hosts several pre-sparsed models along with the sparsification recipe used, and this includes versions of the YOLOv8 detect model. These models are nearly identical in structure to the YOLOv8 pose models, only differing by a few layers in the head, with some parts also differing in parameter count. These recipes could therefore be used for the pose model without any modifications.

The parameters defined in these recipes have been optimized for the YOLOv8 detect model, containing a non-uniform distribution of pruning among the layers. This made it challenging to create comparable recipes for the pose model from scratch. Since the methodology for choosing these parameters is not available, our approach would have had to be experiment-based. Realistically, only a limited number of custom recipes could have been explored within the time frame, potentially limiting the model's results. Although a custom recipe using a uniform pruning distribution was explored briefly, the results seemed inferior and were not explored further.

Due to the challenges associated with creating custom recipes, it was decided that the detect recipes were to be used as a base for all experiments. A consequence of this was that the recipes had to be limited to the sparsity levels already present in SparseZoo. This involved the sparsity levels 50%, 55%, 65%, and 70%, achieved using a magnitude-based method, which were included in the evaluation.

Some modified versions of recipes were also included in the final evaluation. These

Table 3.1: Overview of the models trained with SparseML. The models are denoted with "sml" to indicate their use of SparseML, "p..." to define their sparsity level, and "int8" to specify quantization. The recipes for these models are sourced from SparseZoo, with an "m" denoting any modifications.

Models	Sparsity level (%)	Numeric precision	Note
¢¢¢ sml-base	0	fp32	
sml-int8	0	int8	Quantized base model.
sml-p50	50	fp32	
sml-p50_int8	50	int8	
sml-p55	55	fp32	
sml-p55_int8	55	int8	
sml-p55_int8-m	55	int8	Pose head sparsified.
sml-p65	65	fp32	
sml-p65_int8	65	int8	
sml-p70	70	fp32	
sml-p70_int8	70	int8	
sml-p70_int8-m	70	int8	Pose head sparsified.

included pruning in the pose-specific layers and tried to mimic the pruning distribution present in the original recipes. Pruning and quantization were also evaluated independently to attempt to demonstrate their individual impact on the model. The quantization method used for all quantized recipes was QAT.

The final recipes evaluated are presented in Table 3.1. All models were trained on the COCO pose dataset using consistent hyperparameters, including Ultralytics' default batch size of 16 and a fixed image size of 512x512. However, certain parameters such as epoch and learning rate differ based on the compression techniques being applied.

3.2 GPU evaluation

The GPU experiments were mainly performed utilizing TensorRT. The use of TensorRT was decided to be a fair addition for the comparison since it represents the current state-of-the-art GPU technology in terms of both hardware and software. This choice ensures fairness in the comparison, especially considering that optimization techniques were allowed and evaluated on the CPU side. Without the inclusion of TensorRT, some of the chosen hardware's capabilities would go unutilized and performance would be left on the table. Besides speeding up the base model, TensorRT also includes optimization techniques of its own which will also be taken into consideration.

But even though TensorRT was deemed to be a fair inclusion, an indication of baseline GPU performance was needed for comparison. Therefore, ONNX Runtime was also evaluated and included and served as a GPU base-line when Tensor Cores were not utilized. ONNX was chosen since it generally provides competitive performance on a wide range of hardware.

Table 3.2: Overview of models converted to TensorRT and ONNX formats, indicated by "trt" and "onnx" respectively, followed by the numeric precision.¢

Models	Numeric precision	Note
trt-base	fp32	
trt-fp16	fp16	Applied using PTQ.
trt-int8	int8	Applied using PTQ.
onnx	fp32	Base model in ONNX format.

3.2.1 Hardware

The hardware chosen for the GPU experiments was a NVIDIA Jetson AGX Orin (64GB) running at the 50W (watt) power mode. This was determined to be a suitable option with its inclusion of Tensor Cores and support for current versions of TensorRT. Availability was also a factor. A unit was allotted for doing the experiments, therefore no other hardware was needed to perform the experiments.

3.2.2 Model conversion process

In order to use the TensorRT runtime for the evaluation, the base model had to be converted to a compatible format. This was achieved through a command-line wrapper called trtexec [42], provided by NVIDIA. The model was exported with varying levels of quantization to be included in the comparison.

Similarly, for ONNX Runtime, the base model was converted to ONNX format using Ultralytics. The final model conversions are specified in Table 3.2.

3.3 Validation setup

This section outlines the methods used to evaluate the performance of the models. The evaluation focused on two main metrics: predictive performance and inference time. Given the differing hardware used as well as the variation in optimization techniques applied, a setup that ensured a fair comparison was imperative.

For evaluating the predictive performance, two modified versions of the pose validation function from Ultralytics were created. This validation function was chosen as a starting point as it was known to work as intended and minimized the risk of the results being affected by the implementation. The different output formats of the different runtimes necessitated the creation of these separate versions, one for TensorRT and the other for SparseML. These two versions were used for the evaluation of all models except the one using the ONNX runtime. In the case of ONNX, integration with Ultralytics was already implemented, so no modifications were needed. The COCO pose dataset was used for all validation.

As a result of using the Ultralytics validation function, metrics such as precision, recall, and mean Average Precision (mAP), for both bounding boxes and poses were

obtained. This included both mAP50 and mAP50-95, as described in Section 2.9. These metrics yielded a comprehensive assessment of the models, considering both the model's ability to predict correctly as well as to detect all present instances.

The second validation setup involved measuring the inference speed of the models for the purpose of quantifying speed-ups. This evaluation also required two distinct setups to account for the runtime differences, but these functioned in fundamentally the same way. Time measurements were done using the tqdm python package [43]. All experiments measured only the actual inference time, excluding latency related to pre-processing and post-processing. Warm-up iterations were incorporated into all runs. After the warm-up iterations, the timings of the next 2000 inputs (batch size 1) were collected and averaged to get the final result.

A combined evaluation of these two metrics – predictive performance and inference time – was included to identify models that have been optimized for speed without significant loss of predictive performance. The predictive performance was measured using mAP50 due to its inclusion of both precision and recall, providing an evaluation of the model's performance.

4

Results

The results are presented from two distinct angles to highlight the main aspects considered by the experiments: predictive performance and inference time. The first section will showcase the precision and recall impact of the different optimization techniques evaluated using a variety of setup configurations. Each configuration consists of a unique combination of optimization techniques and compression rates. The second section will showcase the computational speed-up of these configurations of optimization on their respective hardware. The third section will include a combined evaluation of both predictive performance and inference time.

4.1 Predictive performance

This section presents the predictive performance results of the different configurations without considering the hardware. The reason for this is that the model is deterministic given that the weights are fixed. Therefore, the model's behavior is identical across all hardware as long as the model can run on it.

The impact on predictive performance is visualized through different graphs, incorporating the results from validating the models listed in table 3.1 and 3.2. These include the models trained with SparseML as well as those exported with various numerical resolutions using TensorRT.

Figure 4.1 shows the precision for both the bounding box and keypoint predictions for each model respectively. From this, it is clear that the quantized int8 models without pruning (sml-int8 and trt-int8) yield the biggest drop in precision compared to the base model, while the model with half-precision (trt-fp16) shows close to no difference. Among the other models that include both pruning and quantization, sml-p55, sml-p65, and sml-p55-int8 are the ones that stand out. For these models, the precision increases for the box predictions, while there only is a small drop in pose precision when compared to the base case.

A similar trend can be seen across the rest of the graphs in this section. Severe performance drops for models purely quantized to int8 precision on both CPU and GPU while the model quantized to fp16 performs nearly identical to the base model

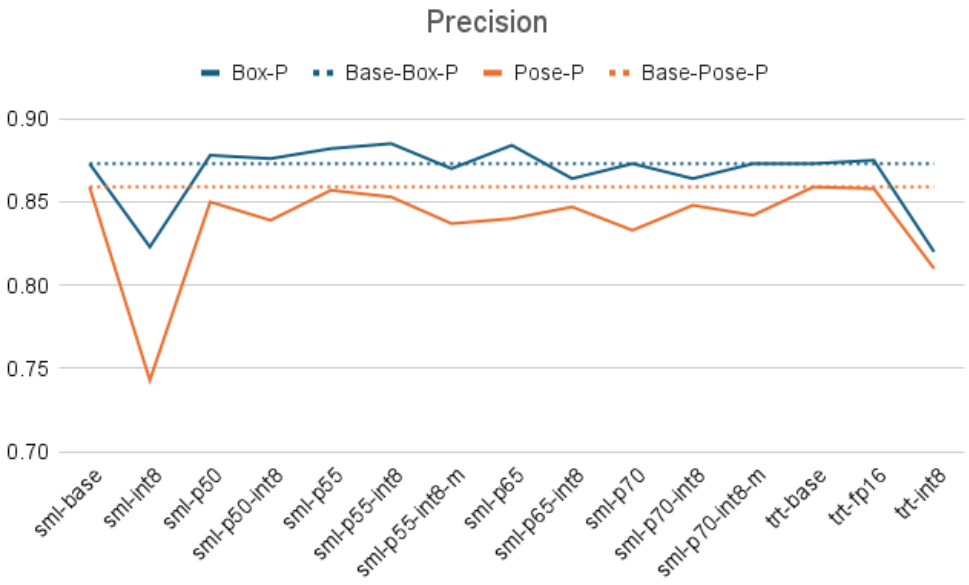

Figure 4.1: Comparison of precision for both pose and box predictions between the optimized models and the base case.

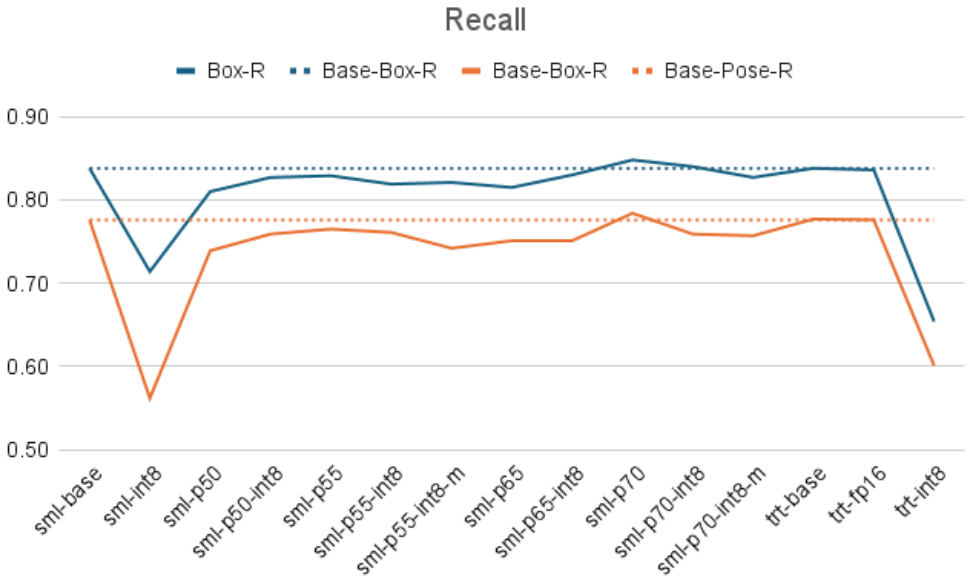

Figure 4.2: Comparison of recall for both pose and box predictions between the optimized models and the base case.

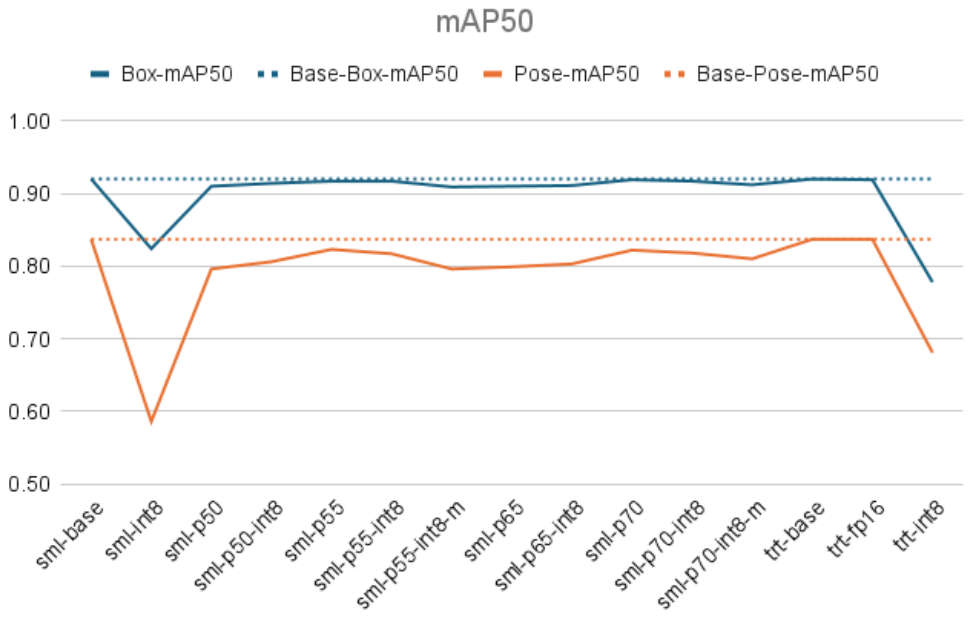

Figure 4.3: Comparison of mAP at an IoU/OKS threshold of 50% for pose and box predictions between the optimized models and the base case.

across the board (Figure 4.2-4.4). The slight performance improvements seen by the sml-p55 and sml-p55-int8 models in the precision plot (Figure 4.1) do not carry over to the other plots. Their performance seems to fluctuate around the baseline, with no single model outperforming the base model in all metrics.

In the recall graph (Figure 4.2), sml-p70 saw a marginal performance improvement while all other models performed slightly below baseline in regards to both pose and box recall.

Figures 4.3 and 4.4 visualize the results of the mAP with a threshold of 0.5 and the average with a threshold between 0.5-0.95, respectively. It is noteworthy that the pruned models, including quantization, are most affected by the uncertain predictions, as seen in Figure 4.4.

An overview of the pose metric results is presented in Table 4.1. Overall, the trt-fp16 model has been the least affected model in comparison to the base model.

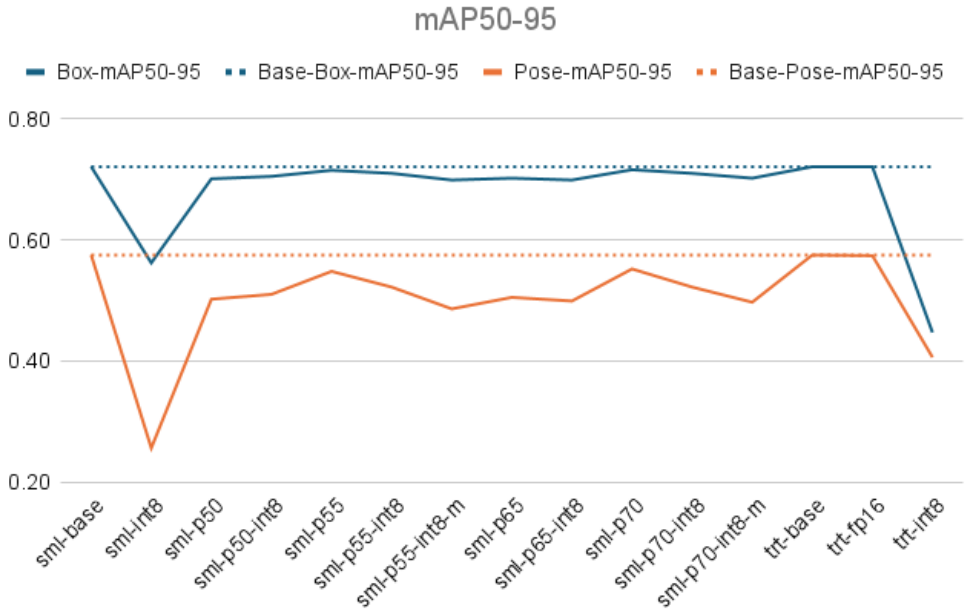

Figure 4.4: Comparison of mAP across IoU/OKS thresholds ranging from 50% to 95% for both pose and box predictions between the optimized models and the base case.

Table 4.1: Overview of the models' predictive performance, including precision (P), recall (R), and mAP for both box and pose predictions.

	Box				Pose			
	P	R	mAP50	mAP50-95	P	R	mAP50	mAP50-95
sml-base	0.873	0.838	**0.920**	**0.721**	**0.859**	0.776*	**0.837**	**0.575**
sml-int8	0.823	0.714	0.824	0.562	0.743	0.562	0.586	0.256
sml-p50	0.878	0.810	0.910	0.701	0.850	0.739	0.796	0.502
sml-p50-int8	0.876	0.827	0.914	0.705	0.839	0.759	0.806	0.510
sml-p55	0.882	0.829	0.917	0.715	0.857	0.765	0.823	0.548
sml-p55-int8	**0.885**	0.819	0.917	0.710	0.853	0.761	0.817	0.522
sml-p55-int8-m	0.870	0.821	0.909	0.699	0.837	0.742	0.796	0.486
sml-p65	0.884	0.815	0.910	0.702	0.840	0.751	0.799	0.505
sml-p65-int8	0.864	0.830	0.911	0.699	0.847	0.751	0.803	0.499
sml-p70	0.873	**0.848**	0.919	0.716	0.833	**0.784**	0.822	0.552
sml-p70-int8	0.864	0.840	0.917	0.710	0.848	0.759	0.818	0.522
sml-p70-int8-m	0.873	0.827	0.912	0.702	0.842	0.757	0.810	0.497
trt-base	0.873	0.838	**0.920**	**0.721**	**0.859**	0.777*	**0.837**	**0.575**
trt-fp16	0.875	0.836	0.919	**0.721**	0.858	0.776	**0.837**	0.574
trt-int8	0.820	0.654	0.778	0.447	0.810	0.601	0.681	0.406
onnx	0.873	0.838	**0.920**	**0.721**	**0.859**	0.776*	**0.837**	**0.575**

* The base cases differs slightly, likely due to variations in number conversions across different runtimes.

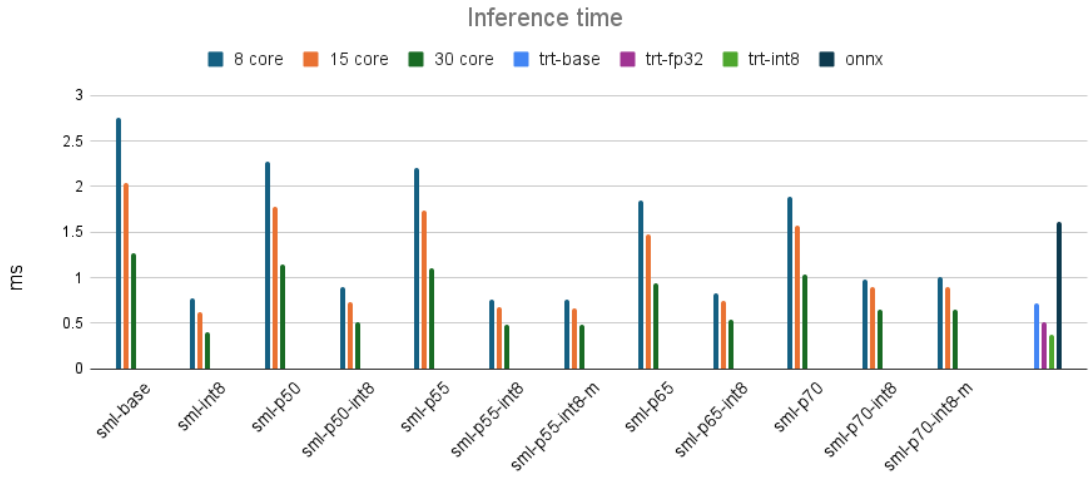

Figure 4.5: Comparison of the models' inference times across different hardware and core configurations. The SparseML models are evaluated on an AMD Genoa with 8-, 15-, and 30-core configurations, while the inference times for TensorRT and ONNX Runtime are evaluated on a NVIDIA Jetson AGX Orin.

4.2 Inference time

The inference time is evaluated across various CPU hardware configurations to compare performance against the selected GPU hardware, NVIDIA Jetson AGX Orin. These tests involve the SparseML models listed in Table 3.1 for the CPU tests, and the TensorRT models listed in Table 3.2 for testing on the GPU.

Figure 4.5 shows the inference time of one item/image using AMD Genoa instances on Google Cloud, comparing 8-core, 15-core, and 30-core configurations. The results show improvements in inference time across all optimizations applied, with the quantized models demonstrating the most significant improvement. The exact results are presented in Table 4.2. Overall, the sml-p55-int8 yielded the best inference times for the 8-core setup, while the sml-int8 models performed best for both the 15-core and 30-core configurations. It is worth noting that the modified versions, which include a sparsified head, perform approximately the same as the versions without modifications.

The inference times running the TensorRT models on GPU are also presented in Figure 4.5. Across all hardware setups, no model outperforms the trt-int8 model. However, running the sml-int8 model on the 30-core configuration resulted in a performance that was exceptionally close (0.028 ms) to that of the trt-int8 model.

4.3 Combined evaluation

This section showcases the combined results of the experiments in order to put the different results into context. This is done using graphs that showcase the relative

Table 4.2: Overview of the inference times in milliseconds across various hardware configurations, including AMD Genoa with 8-, 15-, and 30-core configurations, as well as NVIDIA Jetson AGX Orin (64GB).

	8-core (ms)	15-core (ms)	30-core (ms)	GPU (ms)
sml-base	2.759	2.037	1.269	-
sml-int8	0.778	**0.629**	**0.399**	-
sml-p50	2.272	1.772	1.139	-
sml-p50-int8	0.892	0.728	0.508	-
sml-p55	2.204	1.733	1.106	-
sml-p55-int8	**0.762**	0.671	0.483	-
sml-p55-int8-m	0.765	0.664	0.480	-
sml-p65	1.845	1.471	0.933	-
sml-p65-int8	0.830	0.745	0.537	-
sml-p70	1.893	1.565	1.036	-
sml-p70-int8	0.984	0.894	0.651	-
sml-p70-int8-m	1.007	0.892	0.650	-
trt-base	-	-	-	0.714
trt-fp16	-	-	-	0.509
trt-int8	-	-	-	**0.371**
onnx	-	-	-	1.616

performance vs speed-up of each data point. Due to its poor performance in inference time (see Figure 4.5), onnx was not included in these evaluations.

For these plots, mAP50 was chosen as an approximation of the overall predictive performance of the model, with inference time representing the execution speed. This is helpful to visualize since there is a trade-off associated with model compression. The increases in speed can come at the cost of predictive performance and the relationship between the two is not necessarily linear. The results are visualized in Figure 4.6, with the CPU data based on the highest-end hardware (30 cores).

As can be seen in Figure 4.6, the models optimized using the modified recipes performed nearly the same as their unmodified counterparts. Another notable detail was that the models based on SparseML recipes without quantization were considerably slower than the ones using both pruning and quantization with only a small degradation in mAP50. The un-optimized model running on the CPU was also the slowest as expected. In Figure 4.7, these data points will be excluded as they are not competitive alternatives regarding inference speed. Instead, data from the different CPU hardware will be included to showcase the effect hardware has on performance.

In Figure 4.7, the 30-core CPU at 55% pruning can be seen outperforming the GPU using half-precision in terms of inference speed, at a comparable level of predictive performance. Beyond half-precision, GPU sees a considerable drop in mAP50 and the same can be said for the purely pruned SparseML models.

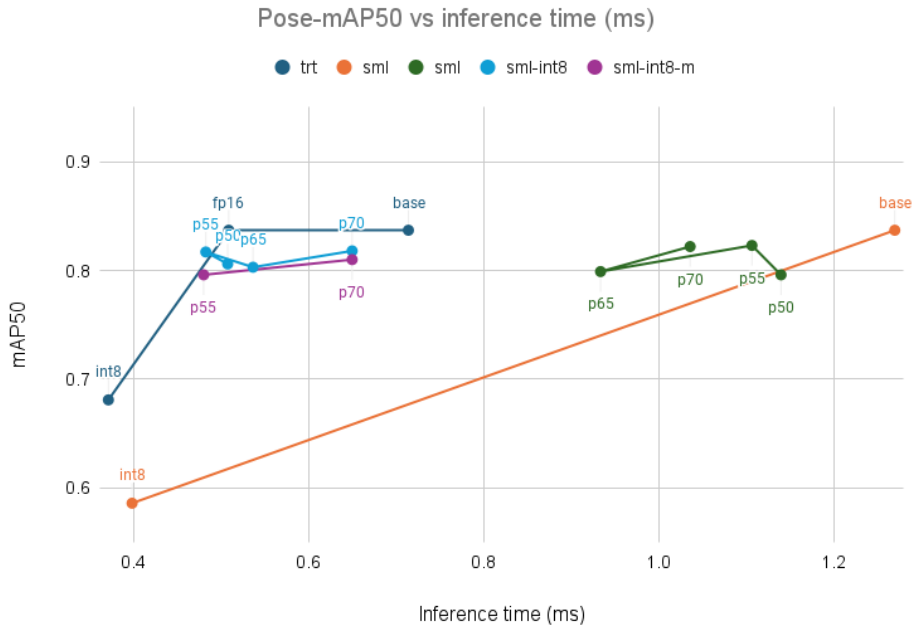

Figure 4.6: Results of pose mAP over inference time for at an OKS threshold of 50%. The SparseML models are evaluated on an AMD Genoa 30-core processor, while the TensorRT models are evaluated on a NVIDIA Jetson AGX Orin.

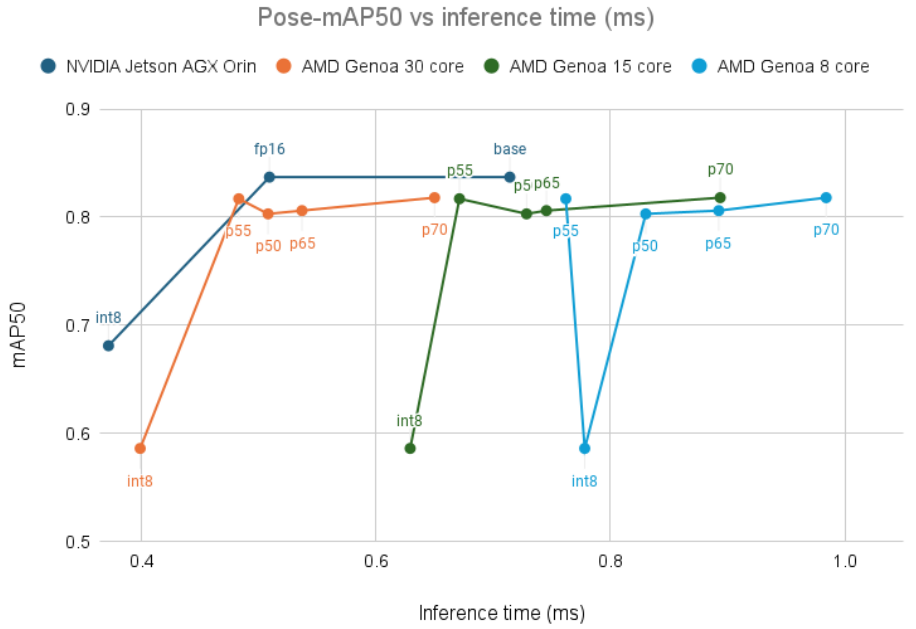

Figure 4.7: Results of pose mAP over inference time at an OKS threshold of 50% across all hardware configurations. This includes the TensorRT models, the quantized SparseML models, as well as the pruned and quantized SparseML models.

5

Discussion

This chapter discusses the findings of the results. It includes an evaluation of critical methodological and background aspects that could have impacted the results, along with an analysis of the results and suggestions for future work.

5.1 Method and background

There are a lot of factors that can influence the effectiveness of compression techniques and this could have had an impact on the results. Since a comprehensive study was not possible due to the time constraints of the thesis, limitations had to be made. The creation of our own implementation would have been interesting to explore but was not feasible due to the time horizon of the project in combination with the small likelihood of achieving superior results. This would also present challenges when it comes to optimized inference. Either, our optimization implementation would have had to be made compatible with DeepSparse or an implementation of our own optimized runtime would have had to be made. This implementation, in turn, would also be limited by time constraints and the low feasibility of achieving superior results.

The adaptation of SparseML for use with pose models also proved to be more challenging than first expected, meaning that the scope of the evaluation had to be reduced considerably. These challenges were a result of the unexpectedly large amount of modifications needed to implement functionality in combination with an undiscovered bug associated with quantization that caused further delays. Without these hurdles, a more comprehensive comparison would have been feasible.

The limited selection of the sparsification recipes could also have had a negative impact on our results. The fact that the base of these recipes was intended for a different task, which was detection, could also have had an affect. The development of new recipes tailored to our model and task could potentially have led to better results, both in terms of predictive performance and inference time. Therefore, the results may not reflect the ideal case.

Other avenues could also have been explored to potentially improve the results. One

of these was the exploration of a wider range of sparsity levels, but the impact likely would have been minimal based on the obtained results. Additional pruning algorithms could also have been evaluated, which potentially could have strengthened the reliability of the results.

GPU implementation was more painless in comparison, although it still required some work. While we did explore built-in optimization methods, they did not yield notable results. However, there may be unexplored methods that could have been included and further evaluated for better results.

5.2 Key findings analysis

The results indicate that the TensorRT model quantized to 16-bit floating point precision (trt-fp16), and the SparseML model with 55% sparsity and quantized with int8 precision (sml-p55-int8) show the most promising results in terms of precision and latency, with sml-p55-int8 being slightly faster. This demonstrates the potential for CPUs to outperform GPUs in certain configurations.

The case of sml-p55-int8 is particularly surprising as it does not have the highest pruning level which goes against previous expectations of higher pruning rates resulting in faster inference. It is difficult to definitively determine the actual reason for this result, as there are many factors that could be at play. These include the compression techniques applied to the models, how DeepSparse leverages them, and how the resulting network structure interacts with the CPU. Our speculations are that a more extensive reduction in model size might lead to decreased computational efficiency, with high computational overhead due to increased sparsity. However, when examining models pruned without quantization, we observe that higher levels of sparsity generally result in reduced inference times, although the SparseML model with 65% sparsity (sml-p65) still outperforms the one with 70% sparsity (sml-p70). This suggests that there may be a threshold beyond which increased sparsity slows down the model. Interestingly, this effect seems more pronounced when quantization is applied.

Another unexpected finding was that the inference time of sml-p55-int8 was faster than that of the base model quantized with int8 precision (sml-int8) on an 8-core CPU setup. This implies that for smaller CPU configurations, there might be greater benefits in speed by combining quantization and pruning. It underscores the importance of considering the selected hardware when determining the compression rates for the model.

Furthermore, our observations revealed a noteworthy trend where models with sparsified heads resulted in a slightly lower mAP50 compared to those without sparsified heads. This outcome may be due to the significantly smaller number of parameters in the head, which could mean a higher concentration of important weights, that have a detrimental impact on the average precision once removed. Additionally, the outcome could have been different if the sparsification recipes were optimized

specifically for our model.

Finding a connection between predictive performance and sparsity level is challenging, as no clear trend can be seen. All models fluctuate around the base-line with no model outperforming the others across the board. Furthermore, the variability in how pruning affects the resulting network structure, especially across different layers, contributes to the observed variation in model performance across different sparsity levels.

A drop in predictive performance was observed across both hardware for the models purely quantized using int8. Since it also occurs with the use of TensorRT, it likely is not a result of compression or implementation, but rather a characteristic of our model. This could potentially be due to a higher sensitivity inherent to the pose task itself.

Another interesting finding is that the TensorRT model with full precision (trt-base) is outperformed by sml-p55-int8 even on a CPU with 15-core configuration. However, given the minimal difference in predictive performance between trt-base and trt-fp16, there appears to be little justification for not using the latter.

Based on our observed results, it is evident that neither NVIDIA nor Neural Magic outperforms the other in all cases of this specific scenario. This brings attention to a new consideration: cost, which could ultimately determine the choice. It is worth noting that the cost of the CPU with a 32-core configuration is approximately equivalent to that of purchasing the NVIDIA Jetson Orin hardware. This makes the 32-core CPU the more expensive choice since the rest of the required components are excluded from its price. If fast inference is not a critical necessity, it could open the door for other, more affordable alternatives where a lower core configuration can be utilized. Otherwise, it may be difficult to justify the process of using SparseML and DeepSparse versus just converting the model to TensorRT, particularly in regards to simplicity.

5.3 Limitations

Due to the limited scope of this thesis, the generality of the results has to be considered. With the experiments being limited to a single model for a single use case, very little can be said about the performance or behavior in other domains. There is nothing to say that the findings would carry over to other network structures since the results could be unique to our situation. However, the results hopefully generalize for CNNs within latency-critical domains, which include a wide range of applications.

The reliability of the evaluation data itself can also be called into question since the results could be heavily tied to the testing conditions of the experiments, with some surrounding factors not being considered. Thermals and power consumption for example were hard to account for due to the limited control over the testing

setup running in the cloud. Therefore, the performance results may not realistically reflect the real-world performance of the model on the given hardware.

Another limiting factor is the dataset is needed during pruning. The dataset used can potentially influence the effectiveness of pruning considerably. In situations without an available dataset, the model would be restricted to the use of one-shot pruning. This could lead to completely different model behaviour, making our results unrepresentative in this scenario.

5.4 Ethical considerations

There are ethical aspects that have to be considered when it comes to the use and deployment of compressed models, specifically for the case of pruning. Since the effects of pruning is not yet fully understood, its utilization could alter the models behaviour in unforeseen ways, resulting in some degree of uncertainty. This can make compressed models unsuitable for mission-critical applications.

In applications like healthcare and automotive, where consequences can be catastrophic, the feasibility of potentially sacrificing average precision and certainty, for the sake of performance should be questioned. At the same time, too low inference performance can be a risk in itself. In some cases, like self-driving cars, slow processing could prove fatal. But in these cases, investing in more capable hardware might be more appropriate, although it may depend on the situation.

It should however be noted that the use of pruning does not always have to result in worse predictive performance and could, in some instances, even improve it.

5.5 Future work

In terms of future work, further investigation into optimizing sparsification recipes could yield valuable insights. Exploring new approaches to designing and fine-tuning these recipes could enhance the efficiency and effectiveness of model optimization techniques. Additionally, evaluating the performance of sparsification recipes across a broader range of hardware configurations and machine learning tasks could provide a more comprehensive understanding of potential use-cases and limitations.

6

Conclusion

This thesis set out to explore if CPUs could be a competitive alternative to modern GPU solutions on a real-time, latency-critical vision task. The vagueness surrounding many of the publicly available performance results in the domain, with many being incomprehensive and not suitable for cross-hardware comparison, was one of the main reasons for this exploration. Despite the limited scope of this work, the findings do suggest that CPUs can be a competitive alternative for low latency machine vision tasks, even at a comparable level of predictive performance. Therefore, when the situation allows for it, CPUs could be considered as a competitive alternative, although in which cases they are preferable remains inconclusive.

References

[1] T. B. Brown, B. Mann, N. Ryder, *et al.*, *Language models are few-shot learners*, 2020. arXiv: 2005.14165 [cs.CL].

[2] NVIDIA Corporation. "Nvidia announces financial results for fourth quarter and fiscal 2024." [Accessed: May 31, 2024]. (Feb. 2024), [Online]. Available: https://investor.nvidia.com/news/press-release-details/2024/NVIDIA-Announces-Financial-Results-for-Fourth-Quarter-and-Fiscal-2024/.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016. DOI: 10.1109/JIOT.2016.2579198.

[4] Y. Lecun, J. Denker, and S. Solla, "Optimal brain damage," vol. 2, Jan. 1989, pp. 598–605.

[5] M. Zhu and S. Gupta, *To prune, or not to prune: Exploring the efficacy of pruning for model compression*, 2017. arXiv: 1710.01878 [stat.ML].

[6] S. Chen and Q. Zhao, "Shallowing deep networks: Layer-wise pruning based on feature representations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 12, pp. 3048–3056, 2019. DOI: 10.1109/TPAMI.2018.2874634.

[7] B. Liu, Y. Cai, Y. Guo, and X. Chen, *Transtailor: Pruning the pre-trained model for improved transfer learning*, 2021. arXiv: 2103.01542 [cs.CV].

[8] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, *Pruning convolutional neural networks for resource efficient inference*, 2017. arXiv: 1611.06440 [cs.LG].

[9] neuralmagic. "Part 1: What is pruning in machine learning?" (2020), [Online]. Available: https://web.archive.org/web/20230130051206/https://neuralmagic.com/blog/pruning-overview/ (visited on 04/05/2024).

[10] Neural Magic, *Sparseml: A library for sparse model training and optimization*, version 1.6.1, Accessed: February 1, 2024. [Online]. Available: https://github.com/neuralmagic/sparseml.

[11] Google Cloud. "Introduction to Cloud TPU." (Accessed: 2024), [Online]. Available: https://cloud.google.com/tpu/docs/intro-to-tpu.

[12] Y. E. Wang, G.-Y. Wei, and D. Brooks, *Benchmarking tpu, gpu, and cpu platforms for deep learning*, 2019. arXiv: 1907.10701 [cs.LG].

[13] NVIDIA Corporation. "Nvidia tensor cores." Accessed: April 18, 2024. (), [Online]. Available: https://www.nvidia.com/en-us/data-center/tensor-cores/. (Accessed: April 18, 2024).

[14] NVIDIA Corporation, *Tensorrt open source software*, version 8.6, Accessed: February 1, 2024. [Online]. Available: https://github.com/NVIDIA/TensorRT.

[15] Neural Magic, *YOLOv8 Detection 10x Faster with DeepSparse: 500 FPS on a CPU*, https://neuralmagic.com/blog/yolov8-detection-10x-faster-with-deepsparse-500-fps-on-a-cpu/, Accessed: May 3, 2024, 2023.

[16] NVIDIA Corporation, *NVIDIA TensorRT*, https://developer.nvidia.com/tensorrt?spm=a2c6h.13046898.publish-article.26.7fe06ffaBIjVNA, Accessed: May 3, 2024, 2024.

[17] NVIDIA Corporation, *Nvidia a100 tensor core gpu*, Accessed: 2024-06-07, 2024. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/.

[18] Neural Magic, *Fastest Software-Delivered AI on CPUs - DeepSparse*, https://neuralmagic.com/deepsparse/, [Accessed 01-02-2024], 2023.

[19] J. Pool, A. Sawarkar, and J. Rodge, *Accelerating inference with sparsity using the nvidia ampere architecture and nvidia tensorrt*, https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/, Accessed: 2024-06-05, 2021.

[20] B. Mehlig, *Machine Learning with Neural Networks: An Introduction for Scientists and Engineers*. Cambridge University Press, Oct. 2021, ISBN: 9781108494939. DOI: 10.1017/9781108860604. [Online]. Available: http://dx.doi.org/10.1017/9781108860604.

[21] E. Fladmark, M. H. Sajjad, and L. B. Justesen, *Exploring the performance of pruning methods in neural networks: An empirical study of the lottery ticket hypothesis*, 2023. arXiv: 2303.15479 [cs.LG].

[22] M. Augasta and T. Kathirvalavakumar, *Open Computer Science*, vol. 3, no. 3, pp. 105–115, 2013. DOI: doi:10.2478/s13537-013-0109-x. [Online]. Available: https://doi.org/10.2478/s13537-013-0109-x.

[23] B. Hassibi and D. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in Neural Information Processing Systems*, S. Hanson, J. Cowan, and C. Giles, Eds., vol. 5, Morgan-Kaufmann, 1992. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1992/file/303ed4c69846ab36c2904d3ba8573050-Paper.pdf.

[24] Z. Yang and H. Zhang, "Comparative analysis of structured pruning and unstructured pruning," in *Frontier Computing*, J. C. Hung, N. Y. Yen, and J.-W. Chang, Eds., Singapore: Springer Nature Singapore, 2022, pp. 882–889, ISBN: 978-981-16-8052-6.

[25] Y. He and L. Xiao, "Structured pruning for deep convolutional neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 5, pp. 2900–2919, May 2024, ISSN: 1939-3539. DOI: 10.1109/tpami.2023.3334614. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2023.3334614.

[26] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, *Pruning filters for efficient convnets*, 2017. arXiv: 1608.08710 [cs.CV].

[27] J. Frankle and M. Carbin, *The lottery ticket hypothesis: Finding sparse, trainable neural networks*, 2019. arXiv: 1803.03635 [cs.LG].

[28] I. Lazarevich, A. Kozlov, and N. Malinin, *Post-training deep neural network pruning via layer-wise calibration*, 2021. arXiv: 2104.15023 [cs.CV].

[29] S. Han, H. Mao, and W. J. Dally, *Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding*, 2016. arXiv: 1510.00149 [cs.CV].

[30] B. Jacob, S. Kligys, B. Chen, *et al.*, *Quantization and training of neural networks for efficient integer-arithmetic-only inference*, 2017. arXiv: 1712.05877 [cs.LG].

[31] Neural Magic, *Sparsezoo: Neural network model repository for highly sparse and sparse-quantized models with matching sparsification recipes*, version 1.7.0, Accessed: April 5, 2024. [Online]. Available: https://github.com/neuralmagic/sparsezoo.

[32] *NVIDIA V100 TENSOR CORE GPU*, NVIDIA, 2020. [Online]. Available: https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf.

[33] NVIDIA Corporation, *Convolutional layers user's guide*, Accessed: 2024-06-11, 2024. [Online]. Available: https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html.

[34] Y. Zhou and K. Yang, "Exploring tensorrt to improve real-time inference for deep learning," in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, 2022, pp. 2011–2018. DOI: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00299.

[35] ONNX, *Onnx: About*, https://onnx.ai/about.html, Accessed: 2024-06-05, 2024.

[36] T.-Y. Lin, M. Maire, S. Belongie, *et al.*, *Microsoft coco: Common objects in context*, 2015. arXiv: 1405.0312 [cs.CV].

[37] F. Chamaki, *Woman selecting packed food on gondola*, [Online; accessed 5-June-2024], 2021. [Online]. Available: https://unsplash.com/photos/woman-selecting-packed-food-on-gondola-YNaSz-E7Qss.

[38] G. Jocher, A. Chaurasia, and J. Qiu, *Ultralytics YOLO*, version 8.0.0, Jan. 2023. [Online]. Available: https://github.com/ultralytics/ultralytics.

[39] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. arXiv: 1506.02640. [Online]. Available: http://arxiv.org/abs/1506.02640.

[40] J. Doe, *Four-person walking near vehicle*, [Online; accessed 5-June-2024], 2021. [Online]. Available: https://unsplash.com/photos/four-person-walking-near-vehicle-H-j7KH1gjP4.

[41] I. Neural Magic, *Deepsparse hardware support*, Accessed: 2024-05-31, 2024. [Online]. Available: https://github.com/neuralmagic/deepsparse/blob/main/docs/user-guide/hardware-support.md.

[42] NVIDIA Corporation, *TensorRT: NVIDIA's Deep Learning Inference Toolkit*, https://github.com/NVIDIA/TensorRT, Accessed: Mars 2024, 2024.

[43] N. Yorav-Raphael and C. da Costa-Luis, *Tqdm: A fast, extensible progress bar for python and cli*, `https://github.com/tqdm/tqdm`, Version 4.65.0, 2024.

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY