THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

EFFECTIVE UTILIZATION OF COMPILE TIME IN COMPUTE GRAPH OPTIMIZATION FOR OBJECT DETECTION

TOMER SEDAN
Fall 2024

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

Mahmut Kandemir
Professor of Computer Science
Thesis Supervisor

Martin Fürer
Professor of Computer Science
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

This thesis explores the current state of graph compilation techniques for machine learning applications. I present a subset of such optimizations that theoretically and empirically have the most striking effects on runtime performance under various conditions. I also propose a novel approach to evaluating optimization effectiveness, and apply it to the selected object detection models. I then discuss how the selection of these techniques changes under different compute paradigms.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I'd like to thank Professor Mahmut Kandemir, whose Compiler Construction course I enjoyed so much it became a thesis. His guidance on this project was indispensable and I am extremely grateful for the opportunities I have had working under him.

I'd also like to thank my good friends on the Pennsylvania State University Advanced Vehicle Team, who lead probably the coolest project on this campus, which I say with absolutely zero bias.

Chapter 1

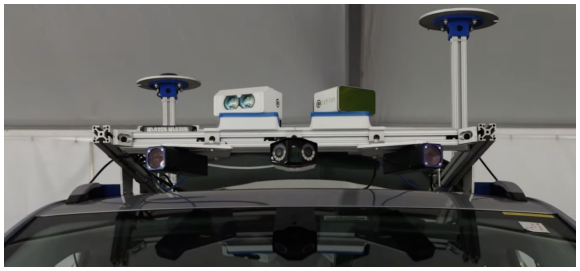
Introduction

1.1 Motivations

As the age of information progresses, it becomes more and more clear that artificial intelligence is one of the most significant technological problems of our time. Not only for the obvious ethical questions it poses (how do we align a machine with "good human moral values"?), but also because current Neural Networks are simply enormous. Modern models (ChatGPT 4 being a prime example) involve hundreds of billions of parameters, a number so large it strains the limits of our current hardware. How do we deal with this exponential increase in compute requirement?

The naive solution is to just keep building bigger computers. It's certainly a valid strategy, and one that corporations often use to scale to millions of users. But equally important are the low-cost, embedded, real-time applications. Consider an autonomous vehicle; compute power is limited not only by the specifications of the battery, but also by the physical amount of space available within the car. And oftentimes, you aren't just running a single model, but several in parallel. Figure 1.1 perfectly exemplifies this basic fact: autonomous cars require many sensors! Each sensor needs its own hardware and software pipelines, to detect objects, localize them, mitigate and filter noise, and to fuse sensor data to generate a map of the environment. Every one of those parts takes up precious cycles on the CPU or GPU.

Figure 1.1: The Pennsylvania State University Advanced Vehicle Team Sensor Suite



In other words, to run important autonomous tasks (such as object recognition and classification, sensor fusion, decision making, controls), available compute hardware needs to be utilized as efficiently as possible. And even for massively-scaled applications that run on the cloud, a meager 10 percent improvement in compute efficiency could save many millions of dollars in server hardware[1].

This is the main motivator for computation graph research. The end goal of such research is to improve the runtime of high performance applications without simply increasing computation power or cost.

As of 2024, the current state of this technology is such that most manufacturers ship platform-specific compilers for their hardware. For example, NVIDIA distributes cuDNN for their graphics cards, whereas Intel ships OpenVINO for its products. Generally, these tools are not built for an end user, and instead act more as intermediate compilers for other libraries to build on top of.

In this paper, we explore ONNX Runtime (ORT), a compiler for the Open Neural Network Exchange model format. ONNX Runtime ships flavors for a few different languages (C#, C++, C, Python and WinML), but we'll be focusing on the most common one, Python, which is a wrapper around the C++ version. ORT can use a variety of backend libraries, called "providers", to actually compile and execute the models. More information about the test methodology is detailed in Section 4.1, but it's important to note that the specific backend evaluated in this paper is cuDNN.

This thesis focuses on evaluating the effectiveness of optimizations that these compilers employ to decrease runtime. It looks at both compilation and subsequent inference times, as well as the difference in efficacy on graphics processors (GPUs) versus central processing units (CPUs). The thesis also proposes one novel quantitative approach to measuring the compiletime-runtime tradeoff, and performs an isolated experiment using that approach. In short, the goal of this paper is to help determine which optimization techniques have the biggest ”bang-for-buck” impacts on object detection model performance.

Chapter 2

Object Detection Models

2.1 Background

A part of what makes compute graph compilation such a highly-researched topic is that there are a practically infinite number of ways to create a model. As such, optimizations that work for a specific type of compute graph may not be effective (or even harmful) to the performance of another.

To obtain meaningful results regarding the efficacy of a certain optimization, we therefore have two choices:

1. Obtain a large sample of models from a variety of applications, and evaluate the optimization on each.

2. Focus on a specific domain, and evaluate the optimization on a highly-used subset of models within that domain.

Since this thesis explores the way optimizations interact with each other, there's a large amount of computation necessary to evaluate optimizations on any given model (see Section 4.1). As such, the domain has been restricted to computer vision applications. Three of the most widely-adopted networks were then selected from within that field. We now describe each model chosen in greater detail.

2.2 YOLOv8

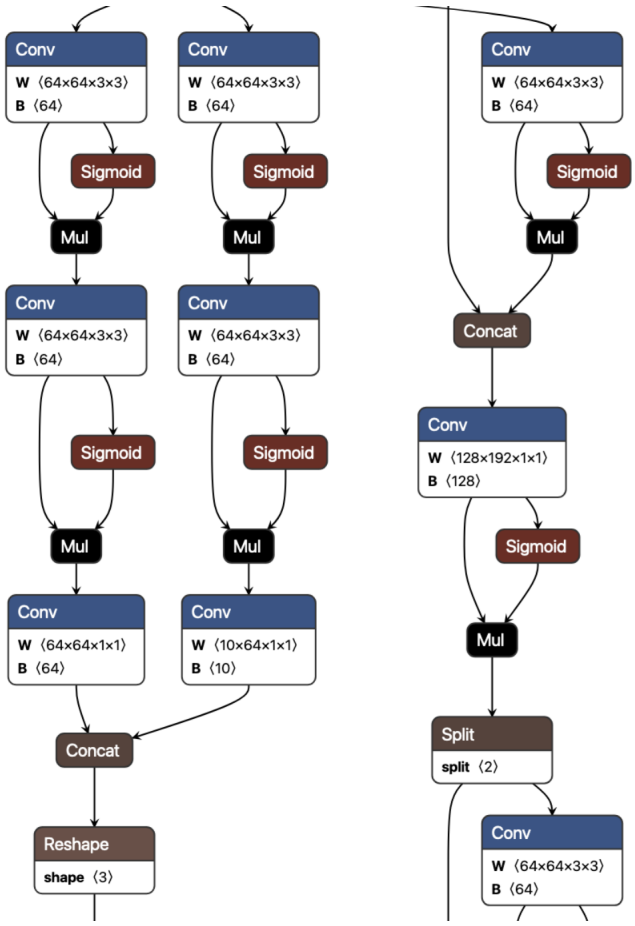
You Only Look Once (YOLO) models are a type of model used to identify the position and classification of objects in a given image frame. As opposed to other model architectures that pursue the highest accuracy by making multiple passes over the input image, YOLO models process the entire image in a single pass, making latency much faster [2]. This makes it a common choice for real-time applications, where the latency of the model is vital to be able to make decisions about the environment tens of times each second.

To give one such example, The Pennsylvania State University Advanced Vehicle Team, which builds self-driving vehicles as part of the SAE AutoDrive Challenge II competition, uses YOLO models almost exclusively in the object detection pipeline. This is largely due to the combination of lower power electronics on a moving, power-constrained vehicle, as well as the high responsiveness requirements of real-time object recognition and localization. In other words, YOLO models are extremely common in the autonomous vehicle industry[2], which is why it's so vital to consider how optimizations affect the runtime performance of this model.

YOLOv8 is the eighth release of the YOLO architecture, and is the model version selected for evaluation here. Even more specifically, the one evaluated in this thesis is a custom-trained YOLOv8 Nano model quantized to 8 bit integers, which is the smallest and fastest available size. Although newer versions have been released since, YOLOv8 (released in 2023) represents a sweet spot between how cutting-edge the architecture is and giving enough time for compilers to iron out any issues.

Figure 2.1 shows part of the "Neck" of YOLOv8, which is the portion of the network that deals with feature extraction. Each node in the graph represents a "kernel", which can be thought of as a function with defined inputs and outputs that is run on a GPU or CPU. The arrows between each

Figure 2.1: YOLOv8 Model Architecture Snippet



node represent the connections between the outputs of the previous nodes and the inputs of the next.

As can be seen from the figure, there are generally three main kernels being utilized in the YOLOv8 architecture. These are convolutions, the sigmoid, and the multiplication operators. We will revisit these in Chapter 3, which deals with the optimizations performed on these graphs. For now, it's simply important to note how much repetition there is in this Figure 2.1; for each connection between a node, there is overhead spent on setting up and running the necessary kernels.

2.3 GoogLeNet

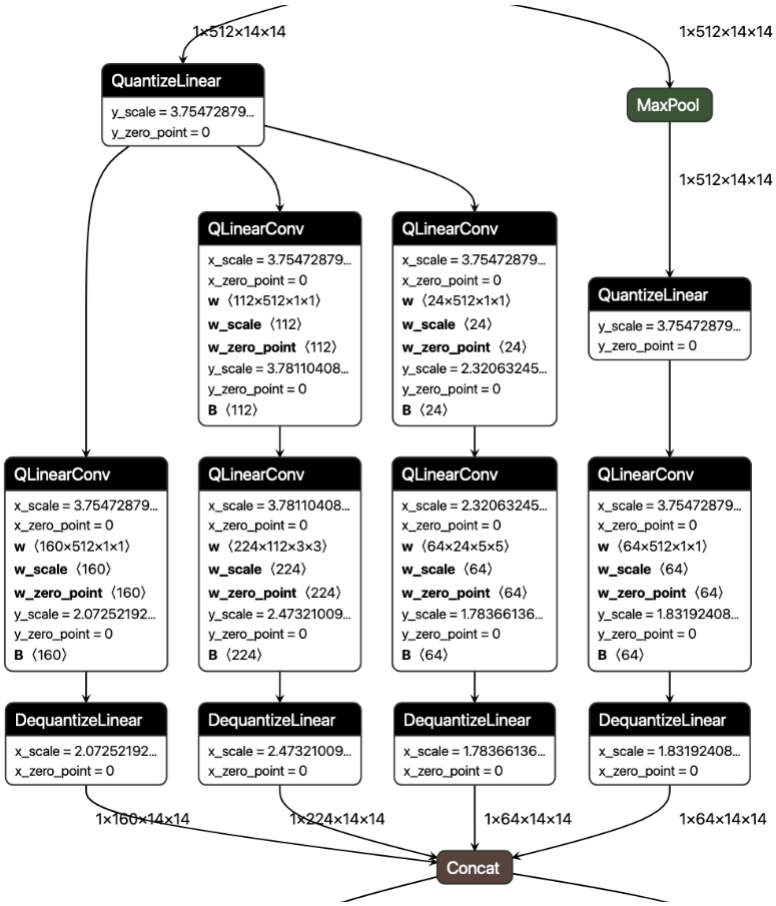
The second model chosen for evaluation is GoogLeNet. GoogLeNet is a convolutional neural network (CNN) architecture primarily used for image classification tasks. One of its defining characteristics is its use of Inception modules, which allow the network to extract features at different scales and depths. Unlike other CNNs, which typically use a single filter size per layer, GoogLeNet processes the input image using multiple filter sizes simultaneously, allowing it to capture both micro and macro details in a single pass through the module [3]. This multi-scale approach leads to a more comprehensive understanding of objects in an image while keeping the computational cost

manageable.

Although it is faster than YOLO, that's largely because YOLO performs an additional task that GoogLeNet doesn't: localization. GoogLeNet can only classify objects (and does so extremely well), but cannot describe where those objects are in a given scene. Somewhat surprisingly, this does not eliminate its usefulness for autonomous applications. Consider a self-driving vehicle that uses LiDAR points to identify regions of interest, and then runs GoogLeNet on a cropped camera frame of that region to identify the object type. This would have a similar result as the YOLO model (albeit requiring additional sensors and a more convoluted pipeline).

GoogLeNet has seen wide application in image recognition tasks, particularly in scenarios where higher accuracy is required but computational resources are constrained. In these contexts, the combination of high accuracy and relatively moderate resource consumption makes it an attractive option for systems that need to process large volumes of images with precision [3]. Moreover, because GoogLeNet can strike a balance between accuracy and efficiency, it remains a popular choice for many image classification tasks across various industries.

Figure 2.2: GoogLeNet Inception Module Structure



GoogLeNet is composed of several stacked Inception modules, as shown in Figure 2.2. Each module combines filters of different sizes (1x1, 3x3, and 5x5) to analyze the input at different levels of granularity. The outputs of these filters are then concatenated, forming the output of the module and feeding it into the next stage of the network.

In Figure 2.2, we can see the flow of information through one such Inception module. Each node in the graph represents a filter or pooling operation, and the arrows indicate the data flow between operations. This parallel processing of the input allows GoogLeNet to efficiently combine multiple perspectives from the same image, while minimizing the computational overhead that would arise from stacking many layers sequentially.

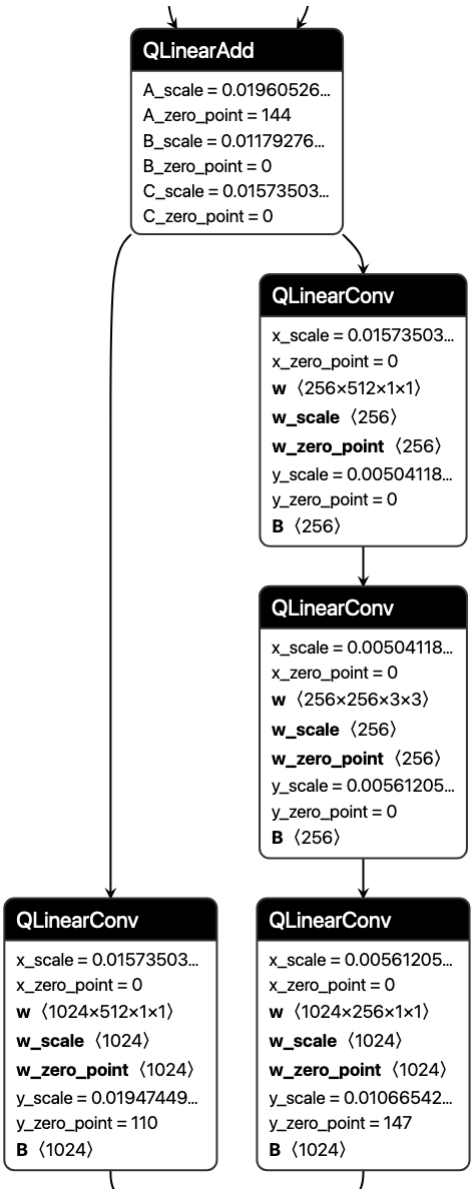
2.4 ResNet

The final model chosen for evaluation in this thesis is ResNet. Residual Networks (ResNet) are a type of deep convolutional neural network (CNN) architecture that introduced the concept of 'skip connections' or 'residual connections.' These connections allow the model to skip over certain layers during training, addressing the vanishing gradient problem that often occurs in very deep networks. By skipping layers, ResNet can train much deeper networks without suffering from the degradation in performance that typically comes with added depth [4]. This innovation allows the model to learn more complex patterns while still retaining the benefits of a deep architecture.

Like GoogLeNet, ResNet is not an image segmentation model but rather an object classification one. ResNet is particularly useful in domains that require the highest accuracy from very deep networks, such as image classification and object detection in large-scale datasets. For example, in satellite imagery, where subtle differences in pixel intensities can indicate important distinctions in terrain or objects, ResNet models are often preferred due to their ability to maintain accuracy even when trained with hundreds of layers [4]. The depth of these networks, coupled with their ability to efficiently train, makes them a highly popular network type for computer vision tasks.

ResNet's architecture is composed of several stacked residual blocks, as illustrated in Figure 2.3. Each residual block contains a series of convolutional layers and skip connections. The key innovation is that instead of directly passing the input through all layers sequentially, ResNet adds the input to the output of certain layers, 'skipping' the intermediate layers. This is shown in the figure as the direct connection from the input to the output of the block.

Figure 2.3: ResNet Block Structure



In Figure 2.3, each block contains two or more layers of convolutions followed by a skip connection. The result of this architecture is that ResNet can effectively train networks with hundreds or even thousands of layers, something that was previously unfeasible due to vanishing gradients and degradation of performance in deep networks.

Now that we have a basic understanding of the structure of these neural networks, we can proceed to discussion of how these networks are optimized.

Chapter 3

Optimization Techniques

3.1 Overview

To optimize compute graphs like those shown in Figures 2.1, 2.2 and 2.3, a variety of tactics are employed by compiler-writers. Figure 3.1 displays a list of all optimizations that ONNX Runtime (ORT) ships with in its C++ (and Python) distributions[1]. In total, there are 59 unique, publicly available optimizations in the ORT compiler, which we can sort into four distinct buckets:

1. 31 optimizations fall under the umbrella of "Kernel Fusion",
2. 10 are a form of "Unnecessary Operation Elimination",
3. 5 perform a "Transform Pass",
4. And the remaining 13 are optimizations specialized on specific kernel types.

Figure 3.1: All Available ONNX Runtime Optimizations

```
optimizations = [
    # https://github.com/microsoft/onnxruntime/blob/b827ab0ef
    # https://github.com/microsoft/onnxruntime/issues/17476
    "AttentionFusion", "BiasDropoutFusion",
    "BiasGeluFusion", "BiasSoftmaxFusion",
    "CastElimination", "CommonSubexpressionElimination",
    "ConcatSliceElimination", "ConstantFolding",
    "ConstantSharing", "ConvAct",
    "ConvAddRelu", "ConvActivationFusion",
    "ConvAddAct", "NhwcFusedConvAct",
    "ConvAddActivationFusion", "ConvAddFusion",
    "ConvBNFusion", "ConvMulFusion",
    "DivMulFusion", "DoubleQDQPairsRemover",
    "EliminateDropout", "DynamicQuantizeMatMulFusion",
    "EmbedLayerNormFusion", "ExpandElimination",
    "FastGeluFusion", "FreeDimensionOverrideTransformer",
    "GatherToSplitFusion", "GatherToSliceFusion",
    "GeluApproximation", "GeluFusion",
    "GemmActivationFusion", "GemmSumFusion",
    "GemmTransposeFusion", "IdenticalChildrenConsolidation",
    "EliminateIdentity", "RemoveDuplicateCastTransformer",
    "IsInfReduceSumFusion", "LayerNormFusion",
    "SimplifiedLayerNormFusion", "MatMulActivationFusion",
    "MatMulAddFusion", "MatMulIntegerToFloatFusion",
    "MatMulScaleFusion", "MatmulTransposeFusion",
    "NchwcTransformer", "NhwcTransformer",
    "NoopElimination", "NotWhereFusion",
    "PreShapeNodeElimination", "PropagateCastOps",
    "QuickGeluFusion", "FuseReluClip",
    "ReshapeFusion", "RocmBlasAltImpl",
    "SkipLayerNormFusion", "EliminateSlice",
    "MemcpyTransformer", "TransposeOptimizer",
    "UnsqueezeElimination",
]
```

Each optimization has its own set of goals and techniques. Kernel Fusion optimizations combine two or more nodes in the computation graph to create one single node that performs the same functional operation. These fused kernels reduce the overhead that comes with setting up memory,

[1] As a side-note, optimization information is unfortunately highly inaccessible as of the time of writing, and I had to dig into ONNX Runtime's source code to individually extract each possible optimization. These practices are hostile to academics!

copying arguments, and spinning up GPU cores on each kernel call. They also often offer some form of algorithmic improvement over the original graph gained by fusion, such as by mathematically simplifying an operation.

Unnecessary Operation Elimination optimizations are among the most intuitive techniques to understand because they come from traditional compilers (such as Clang or GCC). The idea with these optimizations is to find information in the graph that you know ahead of time and exploit it to perform some of the computation at compile-time. For example, if a part of the compute graph involves many consecutive operations involving constant terms, then such a technique may combine all constant terms to reduce work done at runtime.

Transformation Passes are the most complicated optimizations because they usually involve knowledge about the hardware that the model will be executed on. These techniques can reorganize data for better cache effects or exploit specialized processor instructions.

The remaining optimizations are a mixed bag, either not falling directly into one of the other three buckets or falling into more than one. They may look at a specific kernel node and rewrite that kernel in a different form, or may insert some operation to make that kernel executable on a specific hardware platform. This bucket might be thought of as the "supporting role", in the sense that optimizations that fall into this category might not make a significant difference by themselves but instead often enable other, more complicated optimizations.

To explain these all in more detail, we now describe some of the most prominent optimizations, and the way they function. These optimizations were selected as they either present themselves in some interesting form in the results discussed in Chapter 4, or serve as good case studies.

3.2 Quick GELU Fusion

In the vein of "Kernel Fusion" optimizations, Quick GELU Fusion is an optimization that improves the performance of a type of model activation function called GELU. An activation function is essentially a map $f(X) \in \mathbb{R} \rightarrow [0, 1]$ between the real numbers and a range from 0 to 1. The idea of activation functions is to take the output of some operation within a model, and normalize it so that subsequent nodes can work better with it as input. An additional key benefit of these activation functions is that they can turn what would otherwise be a simple linear combination of inputs into a much more complex, nonlinear model [5].

GELU, or Gaussian Error Linear Units, is an activation function with the particular form

$$f(X) = X * \Phi(X)$$

where $\Phi(X)$ is the cumulative density function (CDF) for the Gaussian distribution [6]. This activation function is extremely common in machine learning applications due to its smoothness [5].

However, GELU by itself is difficult to evaluate! The CDF for the Gaussian distribution is

$$\Phi(X) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{X} e^{-t^2/2} dt$$

which would normally be very computationally expensive and perform poorly on massively parallel systems (such as a GPU running a neural network). There are ways to improve performance

by using lookup tables to reduce the amount of repeated computation. Another solution altogether, though, is to simply find another function that approximates GELU.

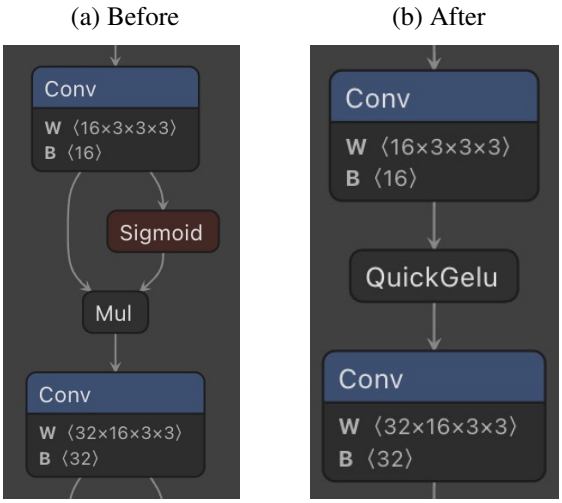
That's Quick GELU. Quick GELU is an implementation of GELU which is defined by

$$f(X) = X * S(\alpha * X)$$

where $S(a) = \frac{1}{1+e^{-a}}$ is the Sigmoid function and α is a distribution parameter (usually set to 1 or 2).

The Quick GELU fusion optimization finds implementations of GELU in the compute graph and fuses them into single Quick GELU operators. Figure 3.2 shows the results of this fusion on the YOLOv8 model selected earlier. The fusion operation removes one whole node with and input and output for each transformation, reducing the amount of memory copying necessary. It also allows a single kernel call to complete the entire activation function without needing to invoke a second kernel, reducing function call overhead on the GPU.

Figure 3.2: Quick GELU Fusion on the YOLOv8 Model



3.3 GEMM-Sum Fusion

GEMM-Sum Fusion is another critical optimization aimed at improving performance in matrix multiplication-heavy operations (a category which neural networks fall squarely into). GEMM, which stands for GEneral Matrix Multiply, is one of the most computationally intensive operations in deep learning models [7]. It computes the product of two matrices, which is a core operation for fully connected layers and many other layers in neural networks.

Now, GEMM kernels are oftentimes followed by a bias addition or another element-wise operation. This sequence of operations creates two distinct computational kernels: one for GEMM and another for the summation. This separation introduces inefficiencies: after the GEMM completes, the output must be copied to memory, then reloaded for the summation step.

GEMM-Sum Fusion addresses this inefficiency by fusing the two operations into a single kernel. Instead of performing the GEMM operation, writing the result to memory, and then loading

it back for the summation, the fused kernel performs both operations in one go. This fusion eliminates the need for intermediate memory writes and reads, significantly reducing memory bandwidth usage.

The fused operation takes the form:

$$Y = A\% * \%B + C$$

where $A\% * \%B$ represents the matrix multiplication, and C is the matrix or vector being added element-wise to the result of the multiplication.

By fusing GEMM with the subsequent summation, the overall kernel execution time is reduced, and function call overhead is minimized [8]. Additionally, this technique enhances cache locality, as the output of the matrix multiplication is immediately available for the summation without needing to be written back to global memory.

3.4 Constant Folding

Next, we can discuss a much simpler optimization: Constant Folding. Constant Folding is a fundamental optimization in the domain of graph compilers that focuses on reducing computational overhead by pre-evaluating expressions involving constants during the compilation process, rather than at runtime [9]. This optimization is particularly beneficial in machine learning models where many operations involve constants, such as parameters or hyperparameters.

In neural networks, constants often appear in various forms, such as fixed scalars in normalization layers, predetermined scaling factors in loss functions, or fixed vectors and matrices. Without constant folding, these operations would be recalculated during every inference or training step, unnecessarily consuming computational resources and time.

The goal of constant folding is simple: it identifies operations in the compute graph that involve only constants and evaluates them once during the graph compilation phase. The result is then hardcoded into the graph as a constant value, eliminating the need for those operations to be performed repeatedly during runtime.

For example, consider an expression like:

$$Y = W * (X + 2)$$

where W and 2 are constants, and X is a variable.

Without constant folding, both the addition and multiplication would be computed during every forward pass. However, with constant folding, the compiler can precompute the expression $W * 2$ during graph construction, and replace the equation with:

$$Y = W * X + W * 2$$

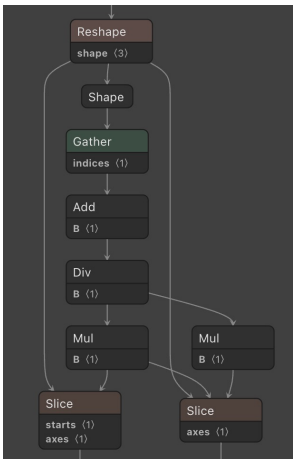
Here, $W * 2$ is computed once, and only $W * X$ remains to be computed at runtime. This reduces the number of operations and improves execution efficiency.

In more complex cases, constant folding can simplify long chains of operations involving constants, further reducing the computational complexity of the graph [9]. This is especially helpful on GPUs, where minimizing unnecessary operations can free up resources for more critical tasks like matrix multiplications or activations.

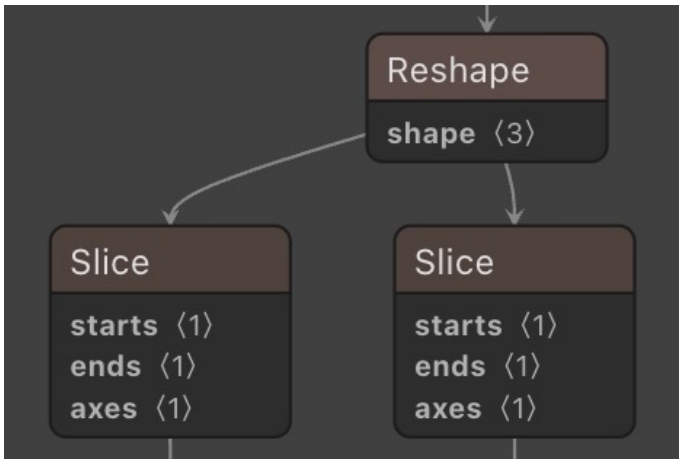
Figure 3.3 illustrates an example of constant folding optimization applied to a compute graph. Before the optimization, several operations with constants are left for runtime evaluation. After constant folding, the graph is simplified, with many constant operations precomputed and replaced with single values.

Figure 3.3: Constant Folding Optimization on the YOLOv8 Model

(a) Before



(b) After



Constant folding not only reduces the number of operations in the graph but also contributes to lower memory usage, as fewer intermediate variables need to be stored during the forward pass. In summary, constant folding transforms a compute graph into a more efficient, streamlined version by eliminating redundant calculations and embedding precomputed constants directly into the graph.

3.5 Expand Elimination

Expand Elimination is an optimization in graph compilers that focuses on reducing unnecessary memory operations when dealing with tensor broadcasting, specifically the ”expand” operation [10]. This optimization is crucial for improving the efficiency of models that frequently perform element-wise operations on tensors of different shapes.

In neural networks, broadcasting allows tensors of different shapes to be combined in operations like addition, multiplication, or comparison, by expanding the smaller tensor to match the size of the larger tensor. Internally, this can result in a memory ”expand” operation where the tensor is conceptually repeated across the dimensions to match the size of the other operand.

The problem with this approach is that the expand operation can introduce unnecessary memory usage and memory copying. Even though the data in the tensor doesn’t change, the operation still creates a larger tensor that occupies more memory, which can slow down execution, especially on memory-bound systems.

Expand Elimination is a graph optimization that avoids this inefficiency. Instead of physically expanding the smaller tensor in memory, this optimization modifies the computation to apply

element-wise operations without the need for actual expansion [10]. In practice, the graph compiler recognizes that the expanded tensor is unnecessary and rewrites the graph to use the original smaller tensor directly, ensuring that the element-wise operation is applied correctly without increasing memory usage.

For instance, in a matrix expression $Y = X + b$, instead of expanding b to match the dimensions of X, the compiler performs the addition in a way that broadcasts b virtually, without expanding its shape in memory. This saves both memory and time by reducing the overhead associated with the expand operation.

The benefits of expand elimination are twofold:

- Reduced Memory Overhead: By avoiding the physical expansion of tensors, the model consumes less memory.
- Improved Execution Time: Eliminating the need for redundant memory operations (like copying data) allows for faster execution of the model.

3.6 Memcpy Transformer

Memcpy Transformer is an optimization technique that targets the inefficiencies caused by excessive or unnecessary memory copying operations. The goal of the Memcpy Transformer optimization is to reduce the overhead associated with these memory copy operations (memcpy) by removing them wherever possible.

In computation graphs, intermediate data between operations is usually stored temporarily in memory and scrapped immediately after kernel startup. For example, after one kernel completes a computation, the result may be copied to a buffer in global memory before the next kernel starts processing it. This constant shuffling of data between different memory locations introduces inefficiencies, as memory copies are relatively slow compared to computation.

The Memcpy Transformer optimization seeks to reduce the number of these redundant or unnecessary memory copy operations by analyzing the compute graph and transforming it in a way that minimizes data movement. There are two main strategies involved in this optimization. One way to reduce memory copying is to allow operations to modify data in place, meaning that instead of creating new memory buffers for intermediate results, the original memory location is updated directly. This reduces the need for memcpy operations between kernels. For example, in a sequence of operations like:

$$Y = g(Z) \;\; Z = f(X)$$

if possible, the Memcpy Transformer will eliminate the need to copy Z to a new buffer before passing it to function g, allowing the same memory space to be reused.

Additionally, in some cases intermediate buffers are used multiple times across different parts of a model, but they don't need to be copied or duplicated in memory. Memcpy Transformer identifies such opportunities where buffers can be shared without causing conflicts. This reduces the number of allocations and memory copies by reusing existing memory buffers rather than allocating new ones for each operation. By transforming the graph to reduce memory transfers, this optimization not only decreases memory bandwidth usage but also improves execution speed, especially in scenarios where the model runs on hardware with limited memory bandwidth or high latency for memory transfers.

Surprisingly, though, the Memcpy Transformer can also *add* copies in certain situations. In my testing, I found that when compiling certain models for CPU instead of GPU, the Memcpy Transformer added extraneous memory copies that were not originally present in the graph.

Figure 3.4: Memcpy Transform on the YOLOv8 Model for CPU

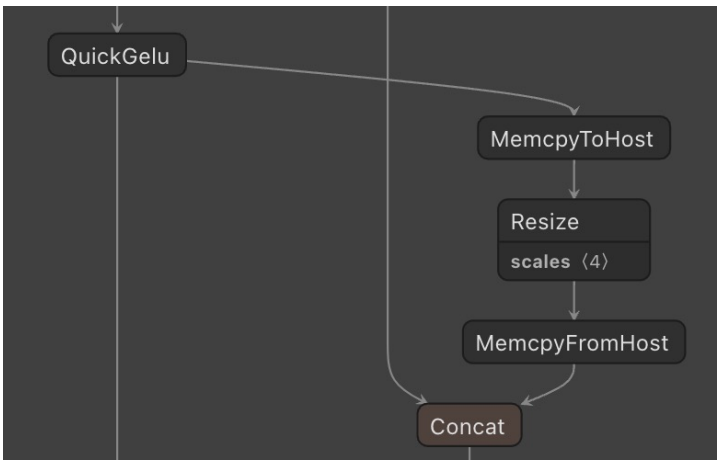


Figure 3.4 shows the result of this transformation. These memory copy operations are inserted in order to bridge the inputs and outputs of nodes when support is lacking [10]. In other words, since GPUs are the primary target of ONNX Runtime, CPU kernels are more limited. Because the kernel options are more limited, the compiler oftentimes cannot specialize on a specific memory layout and needs to perform extra copies to manipulate the data into the input form for each kernel. As would be expected, this results in slower performance, but is also simply necessary unless many variants of each kernel are developed.

3.7 NHWC Transformer

NHWC Transformer is an optimization technique that focuses on improving the performance of tensor-based operations by reorganizing their data layout. NHWC stands for the tensor layout format where the data is stored in the order N (batch size), H (height), W (width), and C (channels) [11]. This format is commonly used in deep learning simply because it's an intuitive way to store data: you have many images you want to process, and each one has a width, height, and color channels.

In contrast, some frameworks or models may default to the NCHW layout, where the order of dimensions is N (batch size), C (channels), H (height), and W (width). While both formats describe the same tensor, the order of dimensions can have a significant impact on performance depending on the underlying hardware and the operations being performed. For instance, some hardware accelerators are optimized for one layout over the other, and using the wrong layout can lead to suboptimal performance due to inefficient memory access patterns.

The NHWC Transformer optimization analyzes the compute graph of a machine learning model and identifies opportunities to convert tensors from the NCHW format to the NHWC format (or other preferred formats) to take advantage of hardware-specific optimizations. This transformation can significantly improve the performance of operations like convolutions, pooling, and other

tensor manipulations that are critical in deep learning. For example, consider a typical convolutional neural network where the input image tensor might initially be in the NCHW format, but the convolutional layers are optimized for the NHWC layout. Without the NHWC Transformer, the data would need to be converted at each layer, introducing unnecessary data transformation steps. With the NHWC Transformer, the data is transformed once, and all subsequent operations can proceed in the optimized layout.

Beyond the NHWC Transformer, eagle eyed readers may have noticed that there is also an NCHWc Transformer available in ONNX Runtime. The function of NCHWc is to further divide the channels dimension into two C dimensions (one big C and one little c) [11]. The idea is that if we can get the final dimension to be exactly some number of elements long, then the compiler can take advantage of specialized ”stride” parameters in a kernel to operate on the data extremely quickly. Using the information that the compiler has about the target hardware, it must determine whether to use NHWC, NCHW, or NCHWc.

Armed with knowledge of the models chosen and some of the optimizations available, we now proceed to evaluation and experimental results.

Chapter 4

Experiments

4.1 Methodology

Now that we have an understanding of the models explored and the key types of optimizations that compilers make, we can describe the experiments performed.

4.1.1 Hardware

First, we need to pick a hardware device. For model inference, a few key factors influence performance: the Graphics Processing Unit (GPU), the Central Processing Unit (CPU), and memory. We need to pick a combination of these three that meets the following requirements:

- Is fairly energy-efficient to simulate low-cost ”edge” applications.
- Has a long enough history to give compilers time to stabilize hardware support.
- Is commonly used in the real world for inference.

To best meet all of these requirements at a reasonable cost, we can employ cloud compute platforms, such as Amazon Web Services (AWS). One of AWS products, Elastic Compute (EC2), enables anyone to ”rent” compute power on an hourly basis. Conveniently, AWS offers a compute tier just for these kinds of applications, named g5. Table 4.1 shows the various flavors of g5.

Instance Size	GPUs	GPU Memory (GiB)	vCPUs	Memory (GiB)
g5.xlarge	1	24	4	16
g5.2xlarge	1	24	8	32
g5.4xlarge	1	24	16	64
g5.8xlarge	1	24	32	128
g5.16xlarge	1	24	64	256
g5.12xlarge	4	96	48	192
g5.24xlarge	4	96	96	384
g5.48xlarge	8	192	192	768

Table 4.1: Amazon Web Services g5 Instance AI-Inference Options

To balance cost and inference speed, the `g5.4xlarge` option was selected. This compute tier includes 16 virtual CPU cores (from an AMD EPYC 7R32 CPU), 64GB of random access memory, and a single NVIDIA A10G GPU with 24GB of GPU memory. Speaking broadly, there are three main manufacturers of non-mobile GPU hardware: Intel, AMD, and NVIDIA. Each manufacturer ships their own dedicated graph compiler, each with its own set of optimizations. To make things simple for evaluation, we focus in this paper on NVIDIA.

4.1.2 Evaluation Algorithm

Back to our initial goal, we seek to determine which optimizations are important for the performance of object detection inference. Naively, one way to do that is to pick a model and a hardware architecture, and to evaluate each optimization individually by turning off all other optimizations.

This would certainly help in a scenario where we had to pick just the single best optimization, but it does not account for the fact that certain techniques may synergize well with others. For example, the Constant Folding optimization described earlier may enable a certain graph pattern to be picked up by a hardware optimization (such as Quick GELU Fusion). As such, it's important to consider as many different combinations of optimizations as possible.

However, simply trying all possible combinations of optimizations is computationally infeasible. ONNX Runtime, our compiler of choice, ships with 59 optimizations out of the box. For each optimization, we can either include it in a given test, or not. If we describe the total number of optimizations as k and the resulting number of possible combinations as n, then we obtain

$$n = 2^k$$

where n for our given $k = 59$ is on the order of 10^{17}. And consider that each combination takes many seconds to run, since we need a large sample size to be confident about the deviation in runtime for any given set of optimizations.

Another idea is to test all combinations in which we only pick 3 or 4 optimizations to enable. The runtime of this method (with respect to a maximum number of selected optimizations c), in contrast to the first, would be

$$n = \binom{k}{c}$$

For $k = 59$ and $c = 3$, that's 32509 combinations to iterate. It's a much more reasonable number, but if we assume each iteration of testing takes about a minute (more on timing later), then the total data-collection time is still on the order of $32509/60/24 \approxeq 23$ straight days. Additionally, we are still not considering that there may be a particularly powerful combination of $c + 1$ optimizations left completely on the table.

To resolve these issues, this thesis uses a technique commonly employed in statistics called "Forward Stepwise Selection". In this method, we iteratively pick the best performing optimization at each step, and then add it to a list of "known good optimizations" that will be enabled by default on the next step. Figure 4.1 demonstrates the basic algorithm used.

This algorithm is essentially ranking all optimizations from best to worst. In the beginning, it loops over all single optimizations, picks the best one, and saves it. Then, it starts with a baseline of the best optimization at that first level, and loops over all remaining optimizations, picks the next best one, and saves that too. This way, we iterate over all optimizations. And if an optimization has particularly good synergy with others, we will be guaranteed to see that interaction in the test, since by the end, all optimizations will be included in the final set.

This method also has the bonus of being much more compute-friendly. At the first iteration, the algorithm performs 59 evaluations, one for each optimization. Then, it picks one, and evaluates $59 - 1 = 58$ optimizations at the next level, and so on. Formalized, this has a time complexity of

$$n = k + (k-1) + (k-2) + \cdots + 1 = \frac{(k+1)k}{2}$$

where the final form was determined as the closed form of a simple arithmetic sum from 1 to k. For a k of 59, we see that this has the ridiculously nice complexity of just $60 * 59/2 = 1770$ iterations. Assuming a 1 minute iteration time, the total computation time is reduced to about $1770/60/24 \approxeq 1.23$ days.

Figure 4.1: Forward Selection Algorithm Pseudocode

```
define forwardSelection:
  start with optimization_level equals 0
  start with previous_best_set equals the empty set

  while optimization_level is less than or equal to k:
    set best_seen_performance equal to 0
    set best_seen_set equal to previous_best_set

    for each optimization "i" not in previous_best_set:
      set performance equal to run_test(previous_best_set + i)
      if performance beats best_seen_performance:
        update best_seen_performance to performance
        update best_seen_set to (previous_best_set + i)

    log the best_seen_set
    set previous_best_set to best_seen_set
    increment optimization_level and repeat
```

4.1.3 Test Statistics

Finally, we describe the testing methodology itself. Each optimization combination results in some change to inference performance for a chosen model. To most accurately measure that change, we must eliminate as many variables as possible.

First, since each model has a different architecture, we are forced to consider models separately. This is not a problem, since we can simply rerun the experiments on each selected model, and compare the final optimization rankings for each.

Next, we consider how to measure inference time. Since all of the models are object detection networks, this is thankfully not a particularly difficult process. Simply put, we can extract individual images from a given test video, resize them to the input tensor size of each model, and record the time it takes to propagate through the network. To obtain a reliable result, this process needs to be repeated hundreds of times for each combination of optimizations. Since we run experiments for both CPU and GPU performance, we set a fairly conservative 302 inference frames per iteration (FPI) for CPU and 604 FPI for GPU.

Inference time, however, is not the only important metric to consider. This paper, after all, includes ”EFFECTIVE UTILIZATION OF COMPILE TIME” in the title. Compile time is an important metric to consider when determining how worthwhile an optimization is to perform. After all, if an optimization is only marginally effective but runs extremely quickly (meaning, compile times are largely undisturbed by its inclusion), then there is no need to omit it from the compilation process.

As such, along with inference testing, we also record the compile time performance for a small number of repeated recompilations ($n = 50$ per iteration). This gives us a general idea of how

compilation speeds are affected by the inclusion or exclusion of a given optimization.

We can also create a novel "composite score" by combining both the compilation and inference metrics. There are a variety of ways to calculate such a composite score, but in this paper, it is simply found as:

$$\texttt{composite\_score} = (\texttt{compilation\_time} * \texttt{inference\_time})^{-1}$$

where we use the inverse to manipulate the score such that a higher score means a better combination (in which both compilation and inference times are minimized). If we then optimize for this composite score (instead of just for inference speed), we can get an idea of at which point the returns on compilation time start to diminish.

In this thesis, we run four experiments, two optimizing for inference time (one on CPU and one on GPU), one for compilation speed, and one for the composite score on GPU. We now detail the results of these experiments.

4.2 Results

4.2.1 GPU Inference

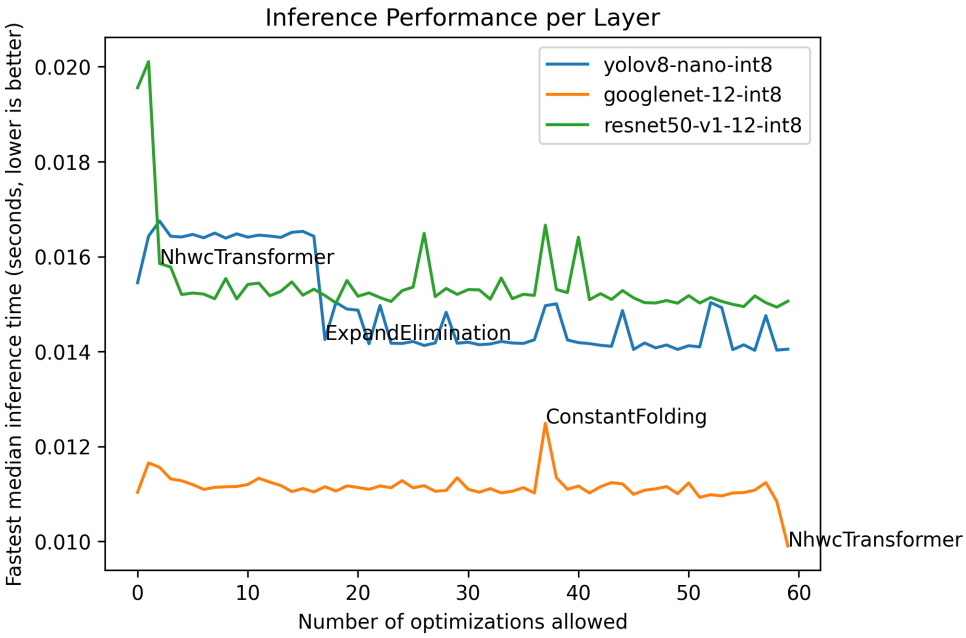
We now describe the results of the first experiment, which seeks to explore the performance of model inference on GPUs. This experiment, as with all the others was run on the g5.4xlarge AWS EC2 instance. For each evaluated optimization combination, 604 unique frames from driving footage (from the Berkeley DeepDrive dataset) were plugged into the model and timed. In this experiment, we compiled all models to run on our A10G NVIDIA data center GPU. The methodology is decribed in further detail in Section 4.1 above.

Figure 4.2 shows the results of the experiment. On the X axis are the number of optimizations allowed, or in other words the "optimization level". For each value on that axis, our algorithm estimates the best score when exactly that number of optimizations out of the total 59 are permitted (i.e. choose the best x optimizations out of the 59 optimizations). On the Y axis is the median inference time of the best scoring combination we found. A lower inference time means a more optimized model. A few points of interest are labeled on the graph, with the corresponding optimization that led to the change in value.

There are a few interesting finds here. We notice that two of the models, GoogLeNet and ResNet, both share large performance increases when the `NHWC Transformer` optimization is applied. As stated in Section 3.7, this is a hardware-specialized optimization that rearranges the tensors to better exploit memory and cache locality as well as data-level parallelism. For ResNet, the change in inference time as a result of this optimization was nearly 20%, whereas it is closer to 10% for GoogLeNet. In YOLOv8, this specific optimization did not have as pronounced of an effect, which may be due to the non-parallel nature of YOLOv8 itself in comparison with GoogLeNet and ResNet (its structure is more serial, as described in Section 2.2).

We also see how `Expand Elimination` as described in Chapter 3 reduced the runtime of YOLOv8 by 15%. Since YOLOv8 is a serial pipeline and does not require the original image more than once, Expand Elimination can remove needless expansions in the operations after each convolution. This reduces the amount of matrix broadcasting required, decreasing memory usage and therefore runtime (since the GPU is generally heavily memory-bound).

Figure 4.2: Inference Experiment Results on A10G GPU



Finally, we note some distinct backwards movements in performance at certain optimization levels across the models. In GoogLeNet particularly, we see a large decrease in performance at the 37th optimization level, and then a subsequent fast return to normalcy. This highlights some of the limitations of this experiment; even with GPU warm up time and a substantial sample size, there are run-to-run variations in inference times that are difficult to account for.

However, we can still proceed by understanding that the algorithm here is selecting the best optimization out of the available options. Since the spikes are so brief, we can conclude that they likely result from run-to-run variance and not from all of the optimizations causing a regression in runtime (otherwise, the effect would last longer than a single optimization). The fact that we see this spike means that the experimental method selected `Constant Folding` as the best possible option, and that all other options performed at least as bad. So we can still conclude that Constant Folding (and the other optimizations selected during those spikes) are likely a net benefit (or at least, does not cause harm) to the runtime of the model.

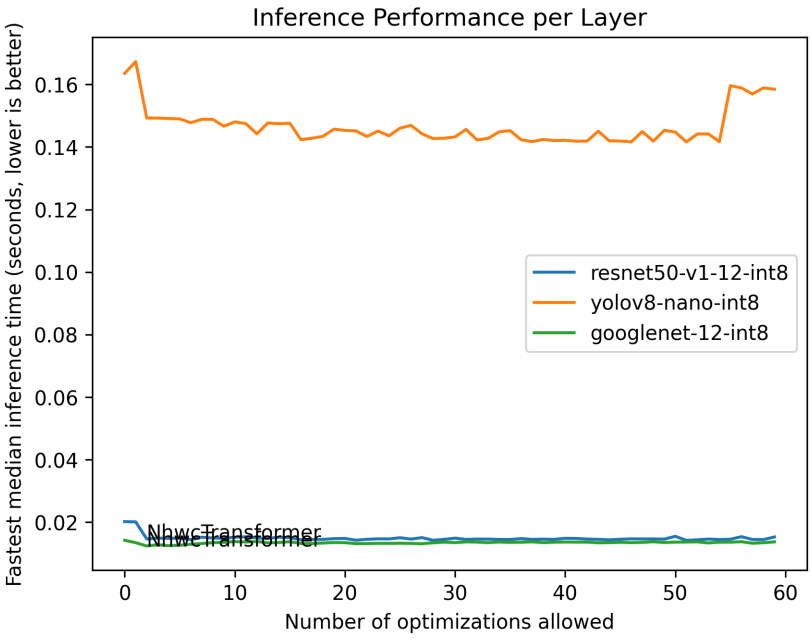
4.2.2 CPU Inference

Next are the results under a similar experiment, but compiled for CPU. The methodology is identical to the previous experiment, except that we ran this test using only 302 frames per iteration instead of 604, since each inference run takes about 2 to 10 times longer on the CPU when compared to GPU.

Figure 4.3 depicts the results in identical form to Figure 4.2. Immediately, we notice that the inference times settle almost immediately and do not budge, with the exception of YOLOv8's final few optimizations. There are a few explanations for why this may be the case.

First, it may be that many optimizations are only considered beneficial for the compute graph if the target architecture is a GPU. For example, many fusion operations might have more dubious

Figure 4.3: Inference Experiment Results on AMD EPYC 7R32 CPU



benefits under CPU, since the cost to call kernels is smaller (we do not have to ”spin up” cores prior to execution as we do on GPU). Another factor in the consistent performance of these models is that they were originally designed to execute on the central processing unit [3][4]. As such, their compute graph architectures are already tailor made to perform well under a CPU compute paradigm. Finally, we might consider that the CPU is so heavily bottlenecked by its own non-ultra-parallelized compute performance that small constant-time compute graph improvements do not result in much difference.

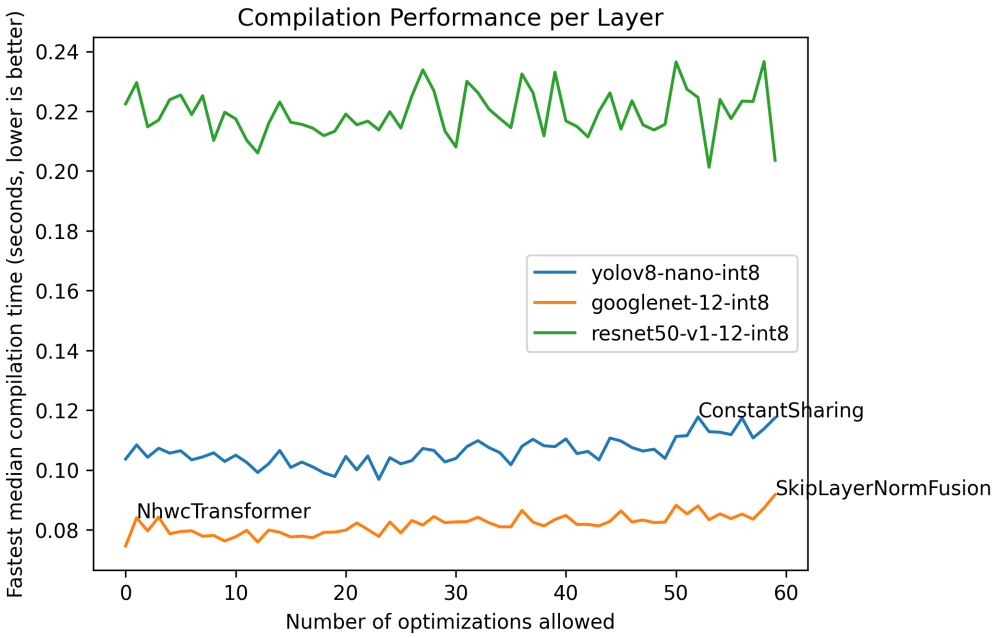
Still, we do note an overall downward trend of a few percent as the optimizations are added, with a larger jump of 8 to 20% in the first few optimizations. Specifically, we see that performance improves quite a bit for both GoogLeNet and ResNet when the `NHWC Transformer` optimization is applied. For YOLOv8, the optimization selected that improved performance the most was `Quick Gelu Fusion`. This optimization fuses those Sigmoid-Multiply operations found in Figure 3.2 and results in a 20-30% reduction in node count, a difference so large it decreases compute time regardless of hardware target.

4.2.3 GPU Compilation

We also performed a GPU evaluation experiment where the cost function minimized at each optimization level was compilation time. The goal of this experiment is to see which optimizations affect compilation performance the most. These tests do not profile inference at all, and simply repeatedly perform re-compilations to evaluate how fast the compiler can optimize each model for the GPU.

Figure 4.4 shows the results of this test. On the whole, we notice that this test has the highest variance of all previous tests, with ResNet struggling particularly to maintain consistent performance between iterations. For YOLOv8 and GoogLeNet, however, the results are more con-

Figure 4.4: Compilation Experiment Results for A10G GPU



sistent, and we've marked some of the points of interest on each line. In particular, we note that `Constant Sharing`, `NHWC Transformer`, and `Skip Layer Norm Fusion` are among the most expensive optimizations to perform.

This makes sense intuitively, since transformation passes that modify data significantly (such as NHWC Transformer) need to make such drastic changes to the compute graph that they heavily impact compilation performance. Constant Sharing and Skip Layer Norm Fusion, on the other hand, suffer the problem of needing to perform significant work in order to determine where the optimization can be applied. Each optimization needs to scan possibly distant and separated sections of the compute graph to find a place where constants may be shared, for example [10].

The overall trend, though, is that there is a slight increase in compilation times as optimizations are added. This increase is not particularly significant on small, one time compilations (around 10-20% compilation performance reduction in the worst case), but would add up if many recompilations needed to occur. For example, in the ONNX Runtime compiler, every time a model is loaded, the compiler automatically performs various optimizations on the model and prepares it for execution on the target hardware [10]. As such, it's vital to reduce that first-compilation latency, since it directly affects how quickly the models can be put on GPU for inference (and therefore how quickly a user can get inference results).

4.2.4 Composite Scores

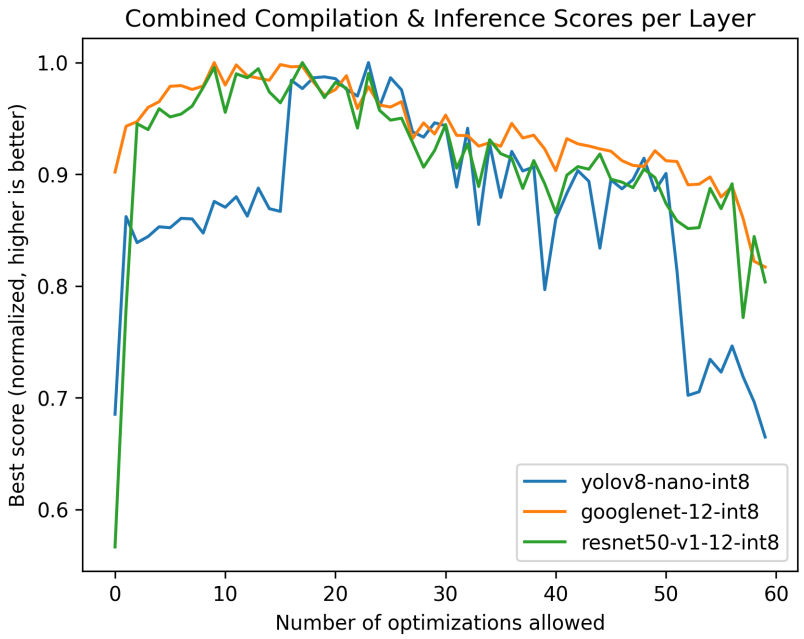
Finally, we display the results of the evaluation when optimizing for the composite compilation and inference score on the selected GPU. As stated earlier in the chapter, this score takes the form:

$$\texttt{composite\_score} = (\texttt{compilation\_time} * \texttt{inference\_time})^{-1}$$

At each step, we pick the combination of optimizations that yields the highest composite score. We run the experiment described in 4.1 for each model, recording the best score at each possible optimization level.

Figure 4.5 shows the normalized scores across each model. Normalization was calculated by finding the largest composite score for each model and dividing each score by that largest score, to get values between 0 and 1. This helps compare the shape of the distribution between the models. It should be noted that normalizing the scores also means we cannot compare models directly to one another, but that is not the goal of this thesis regardless.

Figure 4.5: Composite Compilation and Inference Score Results



There are a few interesting patterns in this data. First, we note that when all optimizations are disabled, the composite score is significantly lower than with even just a single optimization allowed. This suggests that in almost all cases, it is more beneficial to apply at least one very powerful optimization than it is to save on the compilation time and not apply it.

Another observation that can be made is the overall trend for each model type. We notice that for all models, there is a sweet spot in the range of 10 to 25 optimizations at which the performance of the models peaks relative to the amount of time spent on compilation. For all three models, the combined score quickly degrades towards the tail end of the optimizations, suggesting that effort beyond that peak is likely wasted and not necessary to obtain the highest levels of performance.

In the case of YOLOv8, there's a particularly interesting jump in composite scores at the 16-optimization mark. This jump is not intuitive; we would naively expect that if this optimization made such a large difference, that it would have been selected sooner. That, however, is assuming that there are no synergies between optimizations, which we see from this result is far from the case.

This jump was caused by a repeat offender in our results thus far: `QuickGeluFusion`. This optimization, however, does not work effectively unless combined with another: `FuseReluClip`.

FuseReluClip is a support optimization similar to those described in Chapter 3, and enables QuickGeluFusion to function by arranging the compute graph in a standard form [10]. Since FuseReluClip wasn't as beneficial by itself as some of the earlier optimizations, it wasn't selected too early, which is why the sudden jump only occured at level 16. In other words, `FuseReluClip` at optimization level 15 unlocked the full potential of `QuickGeluFusion` at level 16, boosting performance significantly.

Chapter 5

Conclusions and Future Work

5.1 Limitations

We must disclaim that the experiments performed in this thesis were restricted in such a way that limits the extent of the conclusions that can be drawn. First, and most importantly, the sample size of models tested was not particularly large, due to the high amount of compute work required to evaluate a single model (approximately one to two days of raw compute time to run all four experiments on one model using our hardware). Although these models are commonly used in industry, there are a variety of other models, and even different versions of these same models that we did not evaluate on. This is a significant limitation because different models may have drastically different compute graphs, and an optimization which works for one graph may not work for others. As such, the results found in this thesis do not necessarily generalize to all possible object detection models.

Additionally, all experiments were performed on NVIDIA hardware, not accounting for differences that may occur between hardware platforms. For example, an Intel dedicated graphics card may have a particular instruction that can be exploited by a certain optimization better than on the A10G that was tested. It also does not consider implementation differences that may exist between compiler implementations, such as OpenVINO or TensorRT versus cuDNN.

Finally, this paper assumes that all optimizations result in functionally correct models. That is, it does not test whether the models still produce correct output under each optimization. In this case, that is likely a safe assumption to make, as all tested optimizations have been available to the public for several years and have certainly been used on this hardware before.

5.2 Conclusions

In this paper, we explored three different machine learning models: YOLOv8, GoogLeNet, and ResNet. We looked at ONNX Runtime's 59 different optimizations, and evaluated the performance of these optimizations on the various models for GPU and CPU on an AWS EC2 g5.4xlarge instance. In doing so, we learned that the `NHWC Transformer` is the most consistently powerful optimization for this hardware, as it improves the performance of almost all models under both GPU and CPU significantly. We also noticed that this same optimization was one of the slowest during compilation, as it needed to perform many costly tensor transformations on large portions of the computation graph.

Beyond NHWC, the `Expand Elimination` and `Quick Gelu Fusion` optimizations showed promise, particularly in GPU inference. `Quick Gelu Fusion` especially made drastic changes to the YOLOv8 model, but we described how that was specific to models with this type of activation function.

We also noticed that CPU performance did not vary much under heavy optimization, indicating that a small number of highly effective optimizations may be more than sufficient for this use case. Finally, we looked at a holistic measure of "bang-for-buck" performance using a combined compilation and inference score, and noted how scores using this metric peak after around 20 of the 59 optimizations are selected.

As a result, we can conclude that within the constraints of our dataset, three optimizations present themselves as of particular importance: NHWC Transformer, Expand Elimination, and Quick Gelu Fusion. Additionally, for models similar to the three evaluated, a compiler may com-

fortably constrain itself to a smaller subset of around 10 to 20 optimizations and expect to achieve near-optimal performance (in comparison to enabling all optimizations). Of course, a compiler may simply enable all optimizations and achieve a great result, but we have seen through the increased compilation latency that this added work is not always insignificant and may sometimes not result in significantly higher inference performance. Although the list of effective optimizations changes somewhat depending on the model, we may expect that a subset of size $1/3$ of the total number of optimizations can likely be chosen to still contain some of the most consistently powerful optimizations. If this fails, the compiler may choose to specialize on the type of model, selecting optimizations that may apply particularly well for the architecture (such as Quick Gelu for YOLOv8).

5.3 Future Work

As discussed in Section 5.1, this thesis is not comprehensive on the field of object detection models as a whole (a gargantuan task), and instead specializes on a few of the most common models. Although these do give a good idea of what practical performance may look like, additional research is needed to determine what effects these optimizations have on different models.

In that same vein, future work could improve on these findings by generalizing better to the variations in hardware, models and compilers out there. Future work may, for example, explore non-object detection models, or evaluate performance on AMD graphics cards, or perform testing using oneDNN instead of ONNX Runtime.

Additionally, one interesting direction for deeper exploration is the composite score used to correlate compilation and inference time. In this paper, we did not place a preferential weight on inference versus on compilation, meaning that the combined score did not care if we paid a 10% compilation time penalty as long as we also obtained a 10% inference performance gain in return. In the real world, groups may care significantly more about inference than about compilation, and these weightings should be changed to match. Further research might explore how changing these weights affects the optimizations selected or the moment at which adding further optimizations is no longer beneficial.

Bibliography

[1] Nestor Maslej, Loredana Fattorini, Raymond Perrault, Vanessa Parli, Anka Reuel, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, Juan Carlos Niebles, Yoav Shoham, Russell Wald, and Jack Clark. The AI Index 2024 Annual Report. Technical report, AI Index Steering Committee, Institute for Human-Centered AI, Stanford University, April 2024.

[2] Juan Terven, Diana-Margarita Córdova-Esparza, and Julio-Alejandro Romero-González. A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas. *Machine Learning and Knowledge Extraction*, 5(4):1680–1716, November 2023.

[3] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

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

[5] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark, 2022.

[6] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023.

[7] Huaqing Zhang, Xiaolin Cheng, Hui Zang, and Dae Hoon Park. Compiler-level matrix multiplication optimization for deep learning, 2019.

[8] Guibin Wang, YiSong Lin, and Wei Yi. *Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU*. IEEE, 2010.

[9] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991.

[10] Microsoft. ONNX Runtime (v1.19.2). https://github.com/microsoft/onnxruntime, September 2024.

[11] NVIDIA. cuDNN Compiler Documentation Hub – Data Layout Formats, April 2024.

Tomer Sedan

EDUCATION

The Pennsylvania State University **August 2021 - December 2024**

Bachelor of Science in Computer Science, Minors in Statistics and Computer Engineering

Schreyer Honors College scholar, completing honors thesis in graph compiler optimization

Relevant courses: Compiler Construction, Computer Security, Algorithms, Computer Architecture, Microprocessors

Stanford University **June 2022 - August 2022**

CS 161, Design and Analysis of Algorithms, as part of accredited on-campus visit program

TECHNICAL SKILLS

Languages	C++, C, Python, Java, Bash, ARM & x86 ASM, Perl, Verilog, SQL, Javascript
Software & Tools	LLVM, AWS Lambda & EC2, OpenCV, ONNX Runtime, Snowflake, SDL2, ROS 2, Git, Linux, Cython
Knowledge Areas	Optimizing compilers, memory allocators, multithreaded systems, autonomous vehicles, operating systems, embedded microcontrollers, computer vision

WORK EXPERIENCE

Undergraduate Research Assistant, Stanford **May 2024 - Present**

- Combining traditional video compression techniques (H.264) with modern ML-based image compression models using IDR-frame replacement, to obtain lower bitrates at high performance and visual fidelity.
- Working in Professor Tsachy Weissman's lab at Stanford; paper in progress.

Project Management Lead, PSU Advanced Vehicle Team **January 2024 - Present**

- Leading the competition planning and execution efforts to build an L4 self-driving vehicle for SAE's AutoDrive Challenge II. Coordinating and managing work for over 50 students across 6 departments.
- Secured 2nd place autonomous route finish by building an object avoidance system using fused LiDAR.

Sensor Fusion Engineer, PSU Advanced Vehicle Team **August 2023 - January 2024**

- Coordinated the efforts of the Advanced Vehicle Team's Perception department.
- Stitched three camera and two LiDAR sensor feeds to obtain depth and velocity data for objects with a 150 degree field of view, up 67% from the original 90 degrees.

Data Science Intern, Wurl **May 2022 - August 2022**

- Developed a reporting framework, giving 100+ content partners data-driven insight into their advertising performance with Wurl across streamers, channels, and providers.
- Used Snowflake to pull queried data into Tableau, compiling and sending reports using AWS Lambda.

RELEVANT PROJECTS

Optimizing LLVM Backed Compiler **January 2023 - August 2023**

- Built a bespoke compiled systems programming language in C++ using LLVM.
- Architected language features such as type coercion, intrinsics, and a just-in-time compilation mode.

Education-focused Cython Game Engine **October 2021 - February 2023**

- Led creation of an Entity-Component (ECS) game development library aimed towards young students.
- Designed rigid-body physics and a hardware accelerated 2D graphics model in C and Python using SDL.

AWARDS

Evan Pugh Scholar Award, The Pennsylvania State University **February 2024**