FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

BACHELOR THESIS

"High-performance inference on system-on-a-chip devices"

---

to attain the academic degree
Bachelor of Science (B.Sc.)
in Computer Science (Informatik)

FRIEDRICH SCHILLER UNIVERSITY JENA
Faculty of Mathematics and Computer Science

submitted by Vincent Gerlach
Advisor: Prof. Dr. Alexander Breuer
Jena, 18.09.2024

ABSTRACT

The popularity of Deep Learning applications has led to an increased demand for model inference. In response, I accelerate inference using System-on-a-Chip devices to reduce the cost, ensure privacy through locality, and reduce latency. Therefore, I implemented a high-performance convolution for convolutional neural network inference on the CPU using the ResNet50 v1.5 model as an example. My custom implementation is competitive, outperforming a mathematical equal ATen implementation on multiple CPU cores with up to 1.85 faster execution on a fused convolution. Furthermore, I use the ExecuTorch platform to access different hardware and implement an extensible Android app for real-time inference and benchmarking. The Android app allows for inference on the Snapdragon 8 Gen 2 System-on-a-Chip, which includes an Hexagon Tensor Processor for dedicated matrix-matrix multiplications, among other accelerators. With ExecuTorch, the ResNet50 v1.5 model can be lowered to the Hexagon Tensor Processor, enabling a 23.6 speedup over the Snapdragon 8 Gen 2 CPU.

KURZFASSUNG

Die Popularität von Deep Learning Anwendungen hat zu einer erhöhten Nachfrage nach Modell-Inferenz geführt. Als Antwort darauf beschleunige ich die Inferenz auf System-on-a-Chip Geräten, um die Kosten zu senken, die Privatsphäre durch Lokalisierung zu gewährleisten und die Latenz zu reduzieren. Zu diesem Zweck, habe ich eine leistungsfähige Faltung für Convolutional Neural Network Inferenz auf der CPU am Beispiel des ResNet50 v1.5 Modells implementierte. Meine eigene Implementierung ist konkurrenzfähig und übertrifft die ATen-Implementierung auf mehreren CPU-Kernen mit einer bis zu 1,85-fach schnelleren Ausführung. Zusätzlich verwende ich die ExecuTorch Plattform für den Zugriff auf unterschiedliche Hardware und implementiere eine erweiterbare Android App für Echtzeit-Inferenz und Benchmarking. Die Android App ermöglicht, Inferenz auf dem Snapdragon 8 Gen 2 durchzuführen, der neben anderen Beschleunigern auch einen Hexagon Tensor Processor für dedizierte Matrix-Matrix Multiplikationen enthält. Mit ExecuTorch kann das ResNet50 v1.5 Modell auf den Hexagon Tensor Processor übertragen werden, was zu einer 23,6-fachen Leistungssteigerung im Vergleich zur Snapdragon 8 Gen 2 CPU führt.

CONTENTS

1. INTRODUCTION 8
2. BACKGROUND 10
2.1. ResNet50 10
2.1.1. Convolution 10
2.1.2. Shortcut Connections 11
2.1.3. Architecture 12
2.1.4. Theoretical Inputs 13
2.2. Snapdragon 8 Gen 2 14
3. CUSTOM IMPLEMENTATION 16
3.1. Techniques and Implementation 16
3.1.1. Batch Normalization 16
3.1.2. Convolution 17
3.1.3. Max Pooling 20
3.1.4. Global Average Pooling 21
3.1.5. Fully Connected Layer 21
3.2. Performance 22
4. EXECUTORCH 24
4.1. Concept 24
4.2. Backends 25
4.2.1. XNNPACK 26
4.2.2. Vulkan 26
4.2.3. Qualcomm AI Engine Direct 26
4.3. Custom Operator 27
4.4. Android App 27
4.5. Performance 29
4.5.1. Hardware & Quantization 29
4.5.2. Theoretical HTP Performance 32

5. RELATED WORK 34
6. CONCLUSION AND FUTURE WORK 36
REFERENCES 37
LIST OF FIGURES 43
LIST OF TABLES 44
A. RESNET50 V1.5 - NUMBER OF OPERATIONS 45
DECLARATION OF ACADEMIC INTEGRITY 47

LIST OF TERMS

API	Application Programming Interface
AoT	Ahead-of-Time
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DNN	Deep Neural Network
GHz	GigaHerz
GPU	Graphic Processing Unit
HMX	Hexagon Matrix eXtension
HTP	Hexagon Tensor Processor
HVX	Hexagon Vector eXtension
JIT	Just In Time
LLM	Large Language Model
MHz	MegaHerz
ML	Machine Learning
MatMul	matrix-matrix multiplication
NN	Neural Networks
NPU	Neural Processing Unit
PTQ	Post Training Quantization
ReLU	Rectified Linear Unit
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SoC	System-on-a-Chip
TOPS	Tera Operations Per Second
VLIW	Very Long Instruction Word

ACKNOWLEDGEMENTS

I am extremely grateful to my supervisor, Prof. Dr. Alexander Breuer, for helping me with theoretical and technical details and giving me valuable feedback on my thesis. Also, I want to give special thanks to the chair of my committee for supporting me in my work and giving me feedback on my thesis.

Lastly, I am thankful to my family and friends for motivating me and helping me through the process of writing this thesis.

1. INTRODUCTION

In the current decade, we are witnessing the rise of Machine Learning (ML) models such as GPT-4o [40] or GitHub Copilot [24]. As more people use ML to support their workflow, the computational resources needed to perform model inference increase. The term inference describes running a ML model to deduce a prediction from arbitrary input and fixed model parameters. Currently, the inference process for large models is mainly performed on servers [60] provided by large technology companies such as Microsoft and OpenAI and used by millions of people [37]. This results in significant costs for hosting these models, both in terms of hardware and power, which generate significant expenses on the consumer and provider side [30], [41].

System-on-a-Chip (SoC) devices present a cost-effective solution, as most of them are equipped with a Neural Processing Unit (NPU) [47]. An SoC, a computer chip that integrates nearly every component onto a single chip, includes components such as a CPU, Graphic Processing Unit (GPU), memory interface, and input/output interface [59]. This integration significantly enhances power efficiency by reducing physical spacing and embedding dedicated hardware accelerators [13]. An NPU accelerator uses special hard-wired arithmetic units that natively perform matrix-matrix multiplication (MatMul). Thus, they can be utilized for fast and efficient inference, as the computations required for ML models can often be expressed as MatMul operations [22], [23].

NPU-equipped SoCs are often found in newer consumer hardware, allowing models to run efficiently on the consumer device. Mobile devices, especially, are expected to have additional embedded accelerators similar to a NPU to keep power consumption low for ML applications. An example is the Apple iPad Pro with M4, which contains an Apple Neural Engine [27] and a scalable matrix extension for fast MatMul [5], [14]. Another example is the OnePlus 12R [39] smartphone with the Snapdragon 8 Gen 2 SoC [47] with a Qualcomm Hexagon Tensor Processor, which we take a closer look at in Section 2.2. The Qualcomm Hexagon Tensor Processor is further referred to as Hexagon Tensor Processor (HTP).

The evolution of ML on mobile devices is an ongoing development, with the first ML model being used around 2012 to enhance touch input accuracy on mobile devices by mapping the input to the intended touch location [58]. Subsequent ML models were introduced to support new features on mobile devices, such as facial recognition for device login [21] and threat detection for user security [9]. Later, studies were conducted on human activity recognition to enhance the overall user experience [10].

Around 2012, the first frameworks for Deep Learning (DL) appeared, marking a new era for ML with frameworks like Theano [11] and Caffe [29]. A few years later, around 2017, DL frameworks were also available for mobile devices [62]. Unlike previous ML models, DL models can take raw, unprocessed data directly as input, and only the model's architecture is created by experts. For example, Convolutional Neural Networks (CNNs) consist of relatively simple layers using artificial neurons and convolutions [31]. These layers require less manual work and often achieve better accuracy and better generalization to the task [16], [31].

In this thesis, I will focus on the inference of DL models on SoCs. Section 2 provides background about convolutions, the ResNet50 v1.5 model, and the Snapdragon 8 Gen 2, which are the essential parts of this thesis. In Section 3, I explain the methods required to integrate high-performance inference based on a custom implementation of the ResNet50 v1.5 model. Additionally, I compare my custom implementation with an equal implementation through the ATen library. In Section 4, I discuss ExecuTorch as an end-to-end solution to the task of executing inference on SoCs. It combines Ahead-of-Time (AoT) model preparation and execution to take advantage of the different components of a SoC [20]. Further, ExecuTorch tightly integrates with the PyTorch framework, enabling the use of many existing DL models. To take full advantage of ExecuTorch, I implemented an extensible Android app for vision inference with the ExecuTorch runtime covered in Section 4.4. The code for the custom implementation, the ExecuTorch lowered ResNet50 v1.5 model, and the Android app is available at `https://github.com/RivinHD/ImageInference`.

2. BACKGROUND

In this thesis, I use the ResNet50 v1.5 model as an example for CNN to perform inference on a mobile device. A CNN classifies or detects objects inside an image. Thus, it has multiple use cases on mobile devices, such as facial recognition. To port the model to the device, I use ExecuTorch and build an Android app to perform inference and benchmarks. Therefore, we first discuss how a ResNet50 v1.5 works in Section 2.1 and then look at the Snapdragon 8 Gen 2 SoC used on the mobile device in Section 2.2.

2.1. ResNet50 v1.5

The ResNet50 model [25] was another milestone in developing CNNs. It combines the achievements of the early published VGG nets [51] with residual functions. The introduction of residual functions allowed the ResNet50 model to win first place in the ILSVRC 2015 classification task [25]. The ResNet50 model was later improved to ResNet50 v1.5 by NVIDIA [38]. The improvement consists of moving the stride of two in the 3×3 convolutions, whereas the original has the stride two in the first 1×1 convolution.

2.1.1. Convolution

A convolution is a mathematical operation that combines two functions into a new function and is given by the expression:

$$(a * b)(t) := \int_{-\infty}^{\infty} a(\tau) b(t - \tau)\, d\tau$$

Where a and b are the functions being combined. To use it on discrete data, we need to rewrite it into a discrete expression, resulting in the following:

$$(a * b)(n) = \sum_{m=-\infty}^{\infty} a(m) b(n - m)$$

The discrete function is then applied to an input image x and a filter f, which can be seen as a small image, typically between sizes 1×1 and 7×7, holding trainable weights of the CNN. Thus, function a becomes image x, and function b becomes filter f. Further, m expands into the two dimensions, R and S, and n expands to h and w. We can then simplify the discrete function to

$$y_{h,w} = \sum_{r=-R/2}^{R/2} \sum_{s=-S/2}^{S/2} f_{r,s} \cdot x_{h+t,w+s},$$

where y is the output image and h and w are the iterators for the height and width of the selected pixel of the output image. R and S are the height and width dimensions of the used filter. To generalize this expression to multiple input channel C and output channel iterator k, we get

$$y_{k,h,w} = \sum_{c=0}^{C} \sum_{r=-R/2}^{R/2} \sum_{s=-S/2}^{S/2} f_{k,c,r,s} \cdot x_{c,h+r,w+s},$$

where the input channel dimension is based on the input image, and the number of applied filters defines the output channel dimension. Additionally, a stride ς can be added to reduce the output image size so that the new height and width are represented by $h = h_{\text{input}} \div \varsigma$ and $w = w_{\text{input}} \div \varsigma$. Thus, the input image $x_{c,h+r,w+s}$ becomes $x_{c,\varsigma \cdot h+r,\varsigma \cdot w+s}$, but the filter and output image remain unchanged.

2.1.2. Shortcut Connections

The use of shortcut connections comes from the theory behind residual functions, which allowed the ResNet50 to increase its learning capabilities based on network depth. Deep networks have the problem of experiencing higher training and testing errors than those with fewer layers. This problem is known as the degradation problem. It arises from the difficulty of the deeper network to learn an identity mapping that would allow it to achieve at least the same performance as a similar network with fewer layers [25]. Typically, a Neural Networks (NN) learns a hypothesis $H(x)$ through stacked layers based on an input x. This hypothesis also holds for a few stacked layers, i.e., a subset of layers of the original NN, where x is the input of the first layer. The approach of residual functions is to learn $F(x) = H(x) - x$ so that the original functions become $F(x) + x$. The residual approach allows the NN to learn a zero mapping to overcome the degradation problem, i.e., the weights decrease to zero, and the input is added separately. Thus, an approximation of identity mapping is built by the NN, as described in [25]. The zero mapping approach leads us to the implementation of shortcut connections, which do precisely the identity mapping of the input.

We can distinguish between two types of shortcut connections: identity shortcuts and projection shortcuts. The identity shortcut forwards the input to the output of the stacked layers and adds the forwarded input element-wise to the output. The forwarding is possible because the input's dimensions match the output's dimensions. Therefore, we can write an identity projection as $y = F(x, W_i) + x$. Projection shortcuts are needed when the input dimensions do not match the output dimensions. A linear projection is then performed to expand or contract the input dimension as needed. The projection is based on learnable weights W_s, resulting in the equation $y = F(x, W_i) + W_s x$, as shown in [25]. In practice, the projection shortcut is based

on a 1×1 convolution and a batch normalization. The 1×1 convolution is used to reduce the height and width dimensions based on the stride and to expand the channels based on the number of filters used. The 1×1 convolution can only be used because the channel dimension increases, and the height and width dimensions decrease, the more profound the layer is in the architecture.

2.1.3. Architecture

ResNet50 v1.5 [25], [38] is a CNN consisting mainly of concatenated bottleneck blocks. Each bottleneck block consists of a 1×1 convolution, a 3×3 convolution, and a 1×1 convolution with a shortcut from the input of the bottleneck block to its output. A batch normalization and a Rectified Linear Unit (ReLU) activation function follow each convolution of the bottleneck block. A detailed bottleneck block is shown in the legend of Figure 2.1.

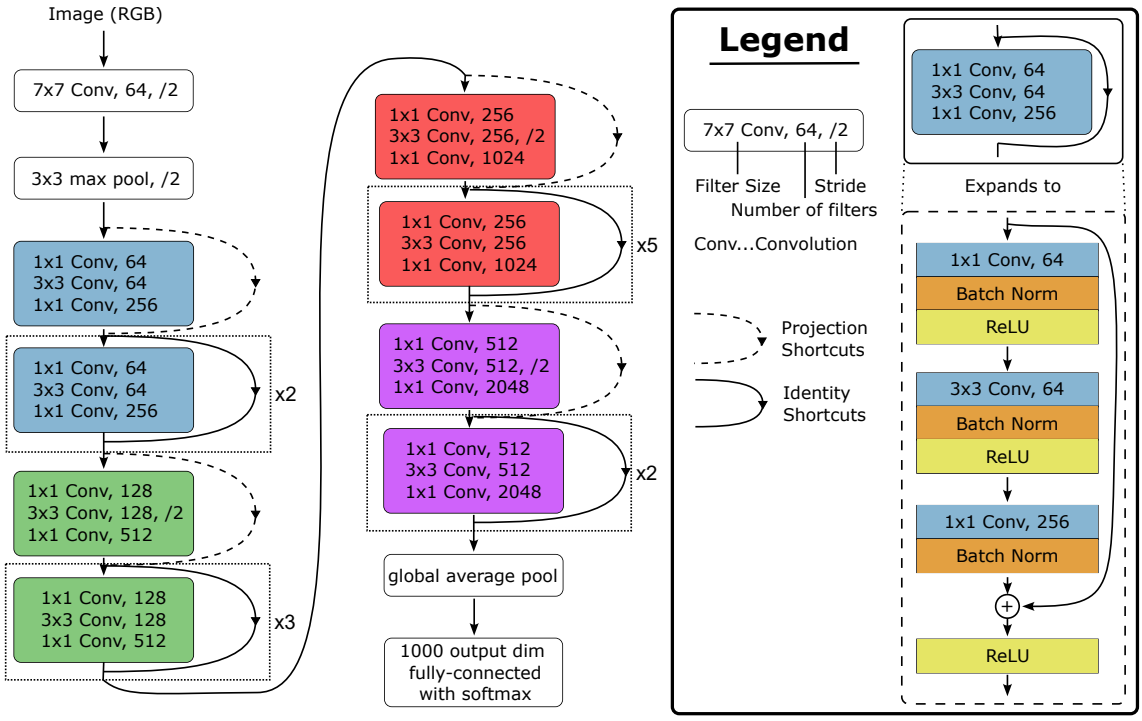


Figure 2.1. The complete architecture of the ResNet50 v1.5. A dotted box with a multiplication number marks repetitions of a bottleneck block with the same shortcut connection.

The model takes a 224×224-pixel image with three color channels as input. The image size of 224×224 is achieved by scaling the image so that the shorter side is 256 pixels and then cropping it to 224 pixels on both sides. The image is further normalized by subtracting the fixed mean of [0.485 0.456 0.406] and dividing the pixel values by the fixed standard deviation of [0.229 0.224 0.225] [53].

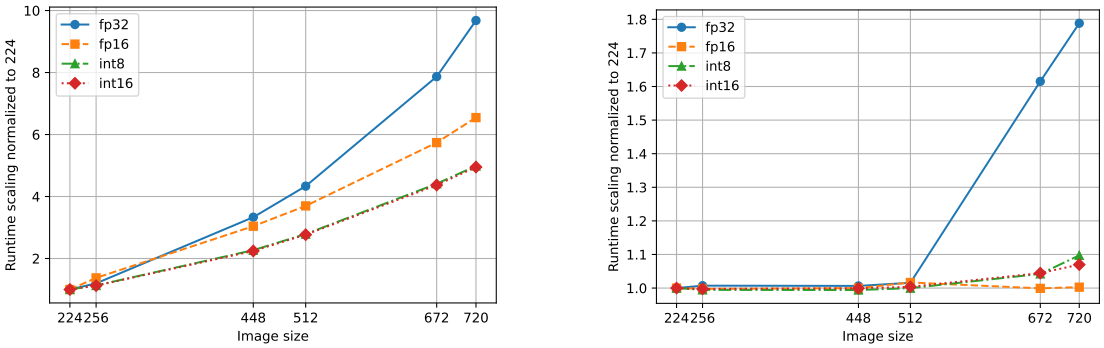
At the beginning of ResNet50 v1.5, a 7×7 convolution with stride two is applied, which halves the size of the image to 112×112. Then, a 3×3 max pooling with stride two is applied, again halving the size to 56×56. Both operations are shown in the first two white blocks of Figure 2.1.

The resulting feature map is then passed to the four blocks consisting of multiple bottleneck blocks, as described at the beginning of this Section 2.1 and indicated by the blue, green, red, and purple colored blocks in Figure 2.1. When a feature map is passed to the second, third, and fourth layer blocks, the first 3×3 convolution halves the size of the feature map by applying a stride of two. In addition, a projection shortcut is used instead of the identity shortcut in each first bottleneck block because the input channel dimension of this bottleneck block does not match the output channel dimension. Therefore, the input channels must be expanded to match the number of output channels. The expansion is done by applying a 1×1 convolution and another batch normalization. Now, we element-wise add the expanded input to the output feature map. An identity shortcut is used for the rest of the bottleneck blocks, which also adds the input of the bottleneck block element-wise to the output. Note that the element-wise addition is performed before the last ReLU of the bottleneck is applied.

After passing through the four blocks, the final stage of the ResNet50 v1.5 model consists of a global average pool and a fully connected layer with an output size of 1000, represented by the last two white blocks in Figure 2.1. The global average pool maps the 7×7 large feature map to a single pixel, where we then pass the channels as a vector of size 2048 to the fully connected layer. Finally, a softmax operation can be applied to transform the output of the fully connected layers into probabilities, i.e., values ranging from 0 to 1. From these 1000 values, each representing a different class in the ImageNet dataset, the human-readable names are recovered and ranked based on the higher value.

2.1.4. Theoretical Inputs

Theoretically, ResNet50 v1.5 could process an input image of any size since there is no mathematical limit to input size for the convolution, max pooling, batch normalization, ReLU, and shortcut operations used. Independence from the image size is ensured by the global average pool in the last step, where the image is scaled down to 1×1 pixel, i.e., to a single value. Increasing the image size should be done carefully, as larger images dramatically increase the computational time and resources needed to perform inference. Figure 2.2 shows two runtime benchmarks of different image sizes where height and width are equal. I chose 224, 448, and 672 for the image sizes because they are multiples of the ResNet50 v1.5 input size. In



(a) Benchmark on CPU (i7-1360P).

(b) Benchmark on GPU (RTX 3070).

Figure 2.2. Benchmark of different input image sizes for the ResNet50 v1.5. Both image dimensions, height and width, are equal. The sizes 224, 448, and 672 are multiples of the input size, and 256, 512, and 720 are standard real-world image sizes.

addition, I also evaluated the image sizes 256, 512, and 720 for reference to standard real-world image sizes. As expected, the runtime in Figure 2.2a scales exponentially with the image size. On the other hand, in Figure 2.2b, the runtime scaling remains the same up to an image size of 512. This behavior can be attributed to the large number of cores a GPU has compared to a CPU. Therefore, a difference in scaling is expected. Also, model results may change and likely become less accurate as the image size deviates more from the expected 224×224 input size, as explained in [35].

2.2. Snapdragon 8 Gen 2

This section provides an overview of the Snapdragon 8 Gen 2 SoC with a focus on the Hexagon Tensor Processor, a component in the Snapdragon 8 Gen 2 SoC. The Snapdragon 8 Gen 2 SoC comprises a Kryo CPU, an Adreno 740 GPU, and a Hexagon processor [48]. The CPU is based on the new Armv9-A architecture [8], which is a 64-bit architecture also known as AArch64 [6].

The Kryo CPU consists of four core types: one prime core, four performance cores, and three efficiency cores [48], [54]. Based on the Cortex-X3, the prime core clocks up to 3.19 GHz. Kryo holds four performance cores with up to 2.8 GHz, divided into two Cortex-A715 and two Cortex-A710. Finally, the CPU comprises three efficiency cores based on the Cortex-A510, which reaches a clock speed of 2.0 GHz.

The Adreno GPU supports the Vulkan 1.3 API, which can also be used for ML purposes.

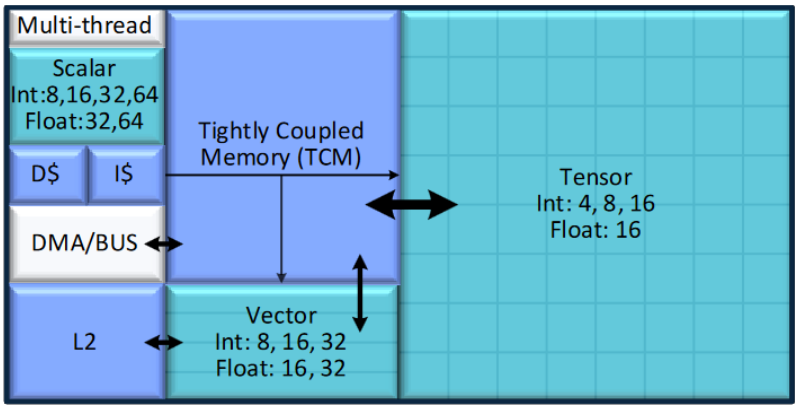


Figure 2.3. Overview of the Hexagon processor with scalar, vector, and tensor coprocessor from [36]

Data type	Composition
Int4	Int4 weights + Int8 activations
Int8	Int8 weights + Int8 activations
Int16	Int8 weights + Int16 activations
FP16	FP16 weights + FP16 activation

Table 2.1. Compositions of data types supported by the Snapdragon 8 Gen 2 HTP [49].

The best efficiency is still achieved by the Hexagon processor, which has dedicated hardware for matrix-matrix multiplication, see Figure 2.3. Unlike the Hexagon processor of the Snapdragon 8 Gen 1, the HTP of the Snapdragon 8 Gen 2 has a dedicated power delivery system that allows it to clock up to 576 MHz independently. The processor has an in-order four-wide Very Long Instruction Word (VLIW) architecture, i.e., four instructions per 128-bit VLIW, and integrates a scalar accelerator, a Hexagon Vector eXtension (HVX), and a Hexagon Matrix eXtension (HMX) [17], [48]. Each instruction packed into a VLIW must be independent to avoid read-after-write and write-after-write hazards. The HTP is a coprocessor of the Hexagon processor and contains specialized matrix multiplication instructions where the output is element-wise added to an accumulator. It performs up to 16 thousand accumulation operations per clock cycle using the 4-bit weights, resulting in a theoretical throughput of 18.8 TOPS at 576 MHz [15], [17]. In addition, the HTP requires quantization in Int4, Int8, Int16, or FP16 to run a ML model [49]. The presented data types are aliases for a composition of data types depending on whether it is attributed to the weight or the activation. The data type composition is shown in Table 2.1.

3. CUSTOM IMPLEMENTATION

This section presents my custom implementation of ResNet50 v1.5 and explains the techniques used to gain performance. It also serves as a methodology for high-performance inference, using ResNet50 v1.5 as an example. Further, we will compare the PyTorch implementation with my own.

My custom implementation is linked to ExecuTorch through the custom operator feature. For details on custom operators, see Section 4.3. The custom operator simplifies the implementation's use, as it can be accessed as a lowered model in my application interface, see Section 4.4. This approach can also be used to integrate existing high-performance implementations of other models into the ExecuTorch runtime.

3.1. Techniques and Implementation

This section outlines the techniques I used to create my custom implementation of ResNet50 v1.5. We discuss the implementation in detail and how I used OpenMP to parallelize it across multiple cores.

3.1.1. Batch Normalization

Batch normalization is a technique that helps speed up training by reducing internal covariance shift [28]. It is calculated by $y = \frac{x-E[x]}{\sqrt{Var[x]+\epsilon}} \cdot \gamma + \beta$, where x is the input, γ is the scaling factor, β adds a shift, and ϵ is a constant for numerical stability. The scaling factor γ and the shift β are parameters optimized during the model's training phase. Furthermore, Var[x] and E[x] are fixed values computed with the training phase and do not change based on the inference input. Therefore, we precompute the factor $\rho = \frac{\gamma}{\sqrt{Var[x]+\epsilon}}$ since it is known before the input is applied, resulting in $y = (x - E[x]) \cdot \rho + \beta$. The pre-computation increases performance if batch normalization is applied more than once since precalculated square root and division save computational resources.

3.1.2. Convolution

Implementing a convolution, explained in Section 2.1.1, can be done in several ways, such as im2col [19] or Winograd [32]. A naive implementation loops over each dimension and processes each kernel independently. This results in ignored spatial locality of cache lines and thus duplicate data loads from main memory, leading to poor performance. Therefore, I block the convolution to process it as MatMul [22]. Note that zero padding is assumed for the convolutions since the dimension of the output tensor is not reduced by the convolution operation.

Blocking of a convolution is achieved by rearranging the input channels and output channels of the convolution into a blocked form. The blocking of the input is obtained by rearranging the channel dimension C, the height H, and the width W like so: $C \times H \times W = (B_C \cdot b_C) \times H \times W \Rightarrow B_C \times H \times W \times b_C$ with B_C being the number of blocks each containing b_C elements. For the filters, we do similarly by rearranging the input channel C, the output channel K, the filter height R and width S, from $K \times C \times R \times S$ into $B_K \times B_C \times R \times S \times b_c \times b_k$, where B_K and B_C are the respective blocks

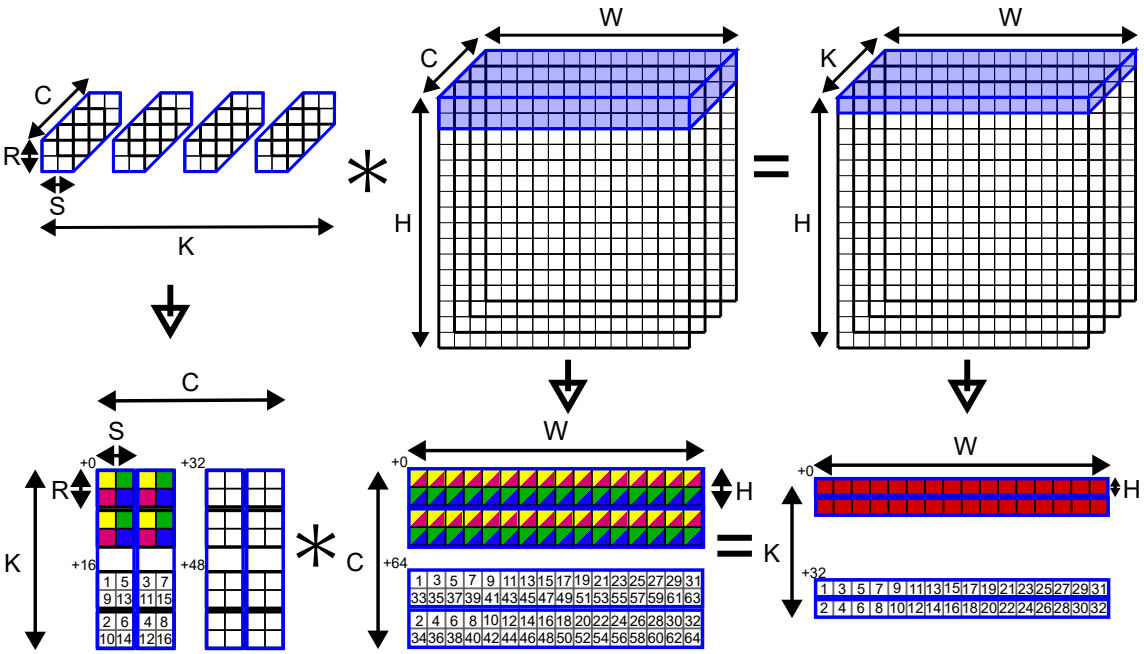


Figure 3.1. Conversion of a convolution into a blocking of a convolution, with blocking along the input channels C and output channels K with block size two. Zero padding is assumed for the shown convolution but is not visualized. The red elements are the output of a single step with blocking of a convolution using the yellow, green, magenta, and blue colored elements as inputs. The numbers inside a block indicate the memory layout of a single block, and the number with the plus at the top left of a block indicates the memory offset of this block.

with the elements b_K and b_C. Last, the memory layout of the output is defined by the blocking $B_K \times H \times W \times b_K$ resulting from $K \times H \times W = (B_K \cdot b_K) \times H \times W$, with B_K being the number of blocks each containing b_K elements. The rearrangement into blocks is visualized in Figure 3.1, where the blue marked part is rearranged into its blocks below. Note that the rearrangement also induces a reordering of the data layout and the loop order for computation so that the highest dimension has a stride of one and is the innermost loop. For example, the input image has the strides $H \cdot W \cdot b_C$ for dimension B_C, $W \cdot b_C$ for dimension H, b_C for dimension W, and one for dimension b_C. Another example is Figure 3.1, where the numbers inside the second blocks indicate the memory layout of a single B_K or B_C block. If one wants to retrieve the memory layout of the whole tensor, one needs to offset the memory layout of each block based on the skipped blocks. In Figure 3.1, the offset is indicated by the number with plus at the top left corner of a block.

The rearranged memory layout allows us to get fast access to the tensor when we stay in the bounce of a single block because of spatial cache locality. Furthermore, the block rearrangement allows us to perform a MatMul with the input and filter, resulting in a blocked output. The MatMul consists of a Filter $\times$ Input = Output, written as dimension we get $(b_k \times b_c) \times (b_c \times W) = (b_k \times W)$. Note that all dimensions are transposed in MatMul compared to previously written data due to a row-major data layout. The process is visualized in Figure 3.1 by the colored elements, where the matching colors are multiplied depending on the C and K channels and summed to two red-marked output rows. When we finish iterating over the input channel blocks B_c, a complete B_k blocked row, marked in red, is finished. Finally, we can apply post-processing to this row before writing it back into memory.

I have divided the convolutions into three types:

1. A convolution that applies batch normalization and ReLU as a post-processing step.
2. A convolution that applies batch normalization, ReLU, and identity shortcut as a post-processing step.
3. A convolution that first processes the projection shortcut and applies it along with batch normalization and ReLU as a post-processing step.

The **first convolution type** can be easily implemented by iterating over B_K, H, B_C, R, S in precisely that order and processing MatMuls inside the loop. Again, the MatMul consists of the dimensions b_K, b_C, and W, so we do not iterate over these dimensions as the MatMul processes them. When the iteration over the dimensions B_C, R, and S are finished, we apply the post-processing steps of batch normalization and ReLU function. The immediate post-processing ensures minimal read and write

operations since the output row is already cached due to the convolution. Note that B_C is processed before H, but logical H should be processed first as H has a smaller stride than B_C. This reordering is done because it allows holding the same complete output rows during the B_C, R, and S loops and immediately post-process. In addition, processing B_C before H has no significant impact if the W and b_c are large enough to fill a complete cache line.

The **second convolution type** performs the same operations, except an additional identity mapping is applied. Therefore, when the iteration over B_C, R, and S are finished, we apply the batch norm to the output row, add the matching row of the identity shortcut, and finally apply the ReLU function.

The **third convolution type** is more complicated than the previous two because it processes two convolutions in the same B_K and H loop. The first convolution is finished by iterating over B_C, R, and S like before. Then, the second convolution computes the projection shortcut by processing a 1×1 convolution in an additional loop over B_C^p, where B_C^p is the input channel dimension of the projection shortcut. The additional 1×1 convolution is needed to get the same dimension as the output. Now, we can apply the post-processing step. First, a batch norm is applied to the output, and another is applied to the output of the projection convolution. Then, both outputs are added together, and finally, the ReLU is performed.

The implementation becomes more complicated because strides greater than one are used by the ResNet50 v1.5. Therefore, we need to adjust the width and height dimension of the output to $W' = W/\text{stride}$ and $H' = H/\text{stride}$, but make sure to calculate the correct index of the input by multiplying the width and height iterator by the stride. The MatMul than becomes $(b_k \times b_c) \times (b_c \times W') = (b_k \times W')$. Since the MatMul processes the width dimension, I set the leading dimension of the input to $W' \cdot \text{stride} = W$. The leading dimension allows us to skip a given amount of elements in this dimension regardless of the given dimension size, so we skip W elements while getting the output for W'. The same method is also applied when the 1×1 convolution of the projection shortcut contains a stride greater than one.

The blocking is set to 32 for both the input and output channel dimensions, as I use the small matrix-matrix multiplication interface of Libxsmm [26]. This blocking can be performed because the ResNet50 v1.5 model consists of channel dimensions multiple of 32, with the lowest being 64. An exception is the first 7×7 convolution, where we have a blocking of the input channel of three because we process the input image with the three color channels.

To further improve the performance of MatMuls, I use the code dispatch interface of Libxsmm [26]. The code dispatch interface sets up a function for the operation and then executes the function with data. The function is set up outside the loops to gain the most performance, and the precalculated function is executed with data inside the loops. To set up the function, the MatMul shape is created based on the output and input matrices, the leading dimension, and the data type. Then, the function is Just In Time (JIT)-compiled from Libxsmm based on the given shape. Finally, the function is executed inside the loops by providing the correct filter, input, and output blocks.

Multicore parallelization is achieved using OpenMP [18]. A collapsed loop parallelizes the two outer loops, B_K and H. In addition, the post-processing step of applying batch normalization, adding the shortcut, and computing the ReLU function is further parallelized by Single Instruction Multiple Data (SIMD) conversion of the innermost loop b_K.

3.1.3. Max Pooling

Max pooling computes the maximum values of input patches, resulting in an output consisting only of those maximum values. Often, a stride of two or more is used to reduce the width and height of the input. In the ResNet50 v1.5 model, a 3×3 max pooling with a stride of two is applied, halving the width and height dimensions of the input.

The implementation of the max pooling can be done similarly to a convolution. The filter weights are replaced by performing a maximum reduction of the inputs and operating independently on the channel, i.e., the channel dimension remains the same. We now follow the loop order B_C, H, W, R, S, b_C, where R and S are the height and width of the max pool kernel. To do a max pool operation, we must first set the output values to the numerically lowest value and then apply the maximum reduction. Therefore, we iterate over B_C, H, W, within which we set the numerical minimum by iterating over b_C and finally apply the maximum reduction by iterating over R, S, and b_C.

Again, OpenMP is used to parallelize across multiple cores. The outer loops, B_C and H, are parallelized by a collapsed loop and further parallelized by SIMD-converting the inner b_C loops since the maximum operation is independent across channels. Loop W is not parallelized because the stride is relatively small, and the values of the next width element are already cached from the previous maximum reduction.

3.1.4. Global Average Pooling

Global average pooling was first introduced by [34] as an alternative to the fully connected layer at the end of a CNN. It is an operation that calculates the average of each channel dimension, thus reducing a feature map from $C \times H \times W$ to $C \times 1 \times 1$. In the ResNet50 v1.5, global average pooling is the penultimate last layer that converts each channel into a single feature, resulting in a feature vector for the fully connected layer. In addition, it keeps the output independent of the input, as explained in Section 2.1.3.

The implementation is relatively simple, as we only need to sum up and average an entire layer. Before iterating over the data, we compute the scale $s = \frac{1}{H \cdot W}$ that will be used to average the data. Then we iterate over B_C, H, W, b_C in precisely that order and sum to the output with the formula y_c += $x_{c,h,w} \cdot$scale, where c, h, and w indicate the iterator over the channel, height and width dimensions, respectively. The scale is applied to each element separately, as it does not degrade the performance much and allows for a better parallel implementation.

OpenMP is again used for parallelization. The two outer loops, B_C and H, are parallelized by a collapsed loop and an additional sum reduction along the output channel dimension. Again, SIMD conversion is used for the inner b_c loop because the sum reduction is channel-independent.

3.1.5. Fully Connected Layer

A fully connected layer is a fundamental building block of a NN. It consists of input nodes connected to every output node, resulting in the formula $y = \alpha(W \times x + b)$, where $x \in \mathbb{R}^n$ is the input and $y \in \mathbb{R}^m$ is the output. $W \in \mathbb{R}^{m \times n}$ are the weights representing the connections between the nodes, $b \in \mathbb{R}^m$ is the bias, and α is the nonlinear activation function.

My implementation is kept simple because it requires significantly fewer flops than the convolutions of ResNet50 v1.5. Therefore, I use the Fastor framework [42] and can write a simple matrix-vector multiplication and an addition to the bias, resulting in b += $W \times x$. The activation function is omitted, as we only needed a linear transformation of the input.

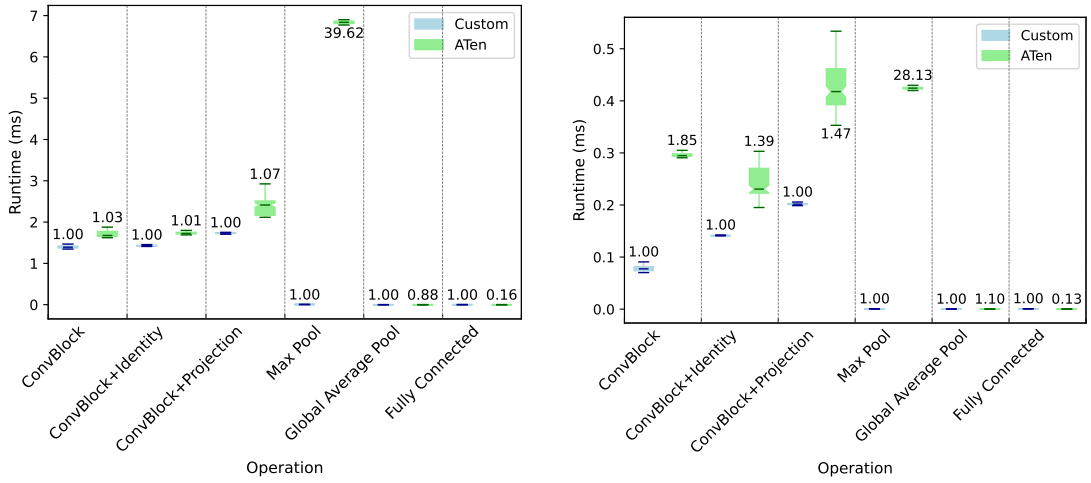
To parallelize the implementation with OpenMP, I block along the columns, since the matrices are in row-major format, and execute the blocking loop in parallel. In my case, the columns are not divisible by the block size, so we need to process the remainder that does not fit into the block.

3.2. Performance

To gain more insight into how effective my implementation is compared to an existing one, we compare it to ATen. ATen is a C++ API and PyTorch's base tensor and mathematical operations library [45].

We compare each operation listed in Section 3.1 to a mathematically equivalent ATen implementation. The benchmark performance results are presented in Figure 3.2. The benchmark was run using the Google benchmark framework with a warmup time of 10 seconds and 100 repetitions of the same benchmark on an Intel i7-1360P CPU. In addition, Intel P-State was set to performance mode on the CPU, and the single-threaded benchmark was fixed to the first core. The benchmarks are run with a block size of 32, a height and width of 224, an input and output channel count of 64, and a kernel size of 3×3. The fully connected layer benchmark is an exception to this setup, as it uses an input size of 2048 and an output size of 1000.

Figure 3.2 shows the benchmark results using a single thread and all 16 hyperthreads. Custom refers to my implementation, which was discussed in this section. The number above each box plot indicates the runtime compared to the custom implementation. "ConvBlock" refers to the operation chain of convolution, batch normalization, and ReLU function, representing the first convolution type in Section 3.1.2. The extension of the "ConvBlock" with "Identity" or "Projection" refers to the second and third convolution type in Section 3.1.2. "Max Pool" refers to the implementation discussed in Section 3.1.3, "Global Average Pool" explained in Section 3.1.4, and



(a) One Thread. (b) All Threads.

Figure 3.2. Performance comparison between my (Custom) and equivalent implementation using ATen on an i7-1360P CPU. The number above each box plot indicates the mean runtime compared to the custom implementation.

finally, "Fully Connected" discussed in Section 3.1.5. Note that the rearrangement of the batch normalization and filter data layout is also done in the benchmark, as the custom ResNet50 v1.5 implementation also performs this.

Figure 3.2a shows that the custom implementation "ConvBlock" can compete with an ATen "ConvBlock" implementation. My max pool operation is significantly faster compared to ATen. The reason why ATen is significantly slower is unknown. The custom global average pool implementation is slightly slower, and the simple implementation of the fully connected layer is significantly slower than ATen. The poor performance of the fully connected layer matters little, as most flops depend on the convolutions in the ResNet50 v1.5 model.

When switching to multiple cores in Figure 3.2b, my three "ConvBlock" operations scale significantly better than ATen. The better scaling may be due to the rearranged memory layout, resulting in a more efficient parallelization. ATen's max pooling operation scales slightly better on multiple cores but is still significantly slower than my custom implementation. My global average pooling operation scales slightly better on multiple cores, outperforming ATen. The custom fully connected implementation is still significantly slower than ATen.

The results should allow us to achieve competitive speeds when applied to the example model. The entire performance of ResNet50 v1.5 is further evaluated in Section 4.5.

4. EXECUTORCH

ExecuTorch[1] is an end-to-end solution that enables on-device inference of PyTorch programs on mobile devices, including iOS, Android, augmented/virtual reality wearables, and microcontrollers. To ensure efficient deployment, it meets three essential requirements [20]:

Portability: Ensure compatibility with a wide range of devices, from high-end mobile phones to microcontrollers and embedded systems.

Productivity: Use the same toolchain for everything: modeling, conversion, debugging, and deployment.

Performance: Deliver a high-performance experience by leveraging CPUs, GPUs, and NPUs.

These ensure that challenges such as power requirements, weak or no Internet connection, and real-time processing are addressed. Please note that ExecuTorch is still in the alpha stage of development at the time of writing. Therefore, concepts and performance may change in the future.

4.1. Concept

The concept of ExecuTorch is based on three steps: exporting, compiling, and executing the model.

First, the model is captured as a graph, representing it as a series of operations such as addition, multiplication, or convolution. Then, we can optionally quantize the model to reduce its memory size and to get better performance, with some loss of accuracy. Quantization enables the model to represent the weights and intermediate tensors using smaller data types, allowing faster computation. Additionally, model quantization can be a hard requirement for some accelerators, e.g., the HTP requires quantization to Int4, Int8, Int16, or FP16 to work.

[1] GitHub repository: https://github.com/pytorch/executorch

Second, the graph is converted to the Core ATen dialect [43], which decomposes thousands of operators into a few fundamental operators. The Core ATen dialect makes implementing these operators easier and applies more fundamental transformations to improve performance. Next, the graph of the Core ATen dialect is lowered to an Edge dialect [44] that contains the Core ATen operators and user-defined custom operators. The Edge dialect has the constraint that everything is represented as a tensor, e.g., scalars become rank zero tensors. In addition, we can lower these graphs to a target-aware backend such as XNNPACK, Vulkan, or Qualcomm AI Engine Direct, which further improves performance or enables the use of specialized hardware such as the HTP. Lowering to a backend is done by swapping the graph with a semantically equivalent graph based on the provided operators of the lowered backend. The lowering process involves partitioning a subgraph, which then gets sent to the backend to be replaced by one or a series of specialized operations. Finally, the lowered graph is converted into an ExecuTorch program. At this stage, we can add optional memory planning for the intermediate tensors to reduce the number of allocations and deallocations. Memory planning is available because it can be planned based on the static graph and is done by algorithms automatically. However, it can be extended or overridden by custom memory plans.

Third, the compiled ExecuTorch program can now be loaded and executed by an AoT-compiled ExecuTorch shared kernel library that contains the ATen operators, target-specific libraries, and custom operators. In Addition, ExecuTorch supports selective build, a feature that allows one to choose only the necessary kernels the program uses, thereby minimizing the binary file size [20].

In this thesis, I use the ResNet50 v1.5, explained in Section 2.1, as an example model for image classification. Therefore, I lowered the model based on the explained concepts and used several programs of the same model using the XNNPACK backend and the Qualcomm AI Engine Direct backend. In addition, I use quantized and initial models for each backend. Further, I use the default memory planning, which greedy optimizes the intermediate tensors by reusing as many allocated tensors as possible.

4.2. Backends

This thesis uses multiple backends to lower to the different hardware available on the Snapdragon 8 Gen 2 chip. The XNNPACK backend is used for the CPU, Vulkan for the GPU, and Qualcomm AI Engine Direct for the HTP on the Qualcomm chip.

4.2.1. XNNPACK

XNNPACK[2] is a highly optimized library for running NN inference on CPUs of various architectures, such as Arm, x86, WebAssembly, and RISC-V. It is an open-source project hosted by Google and is included by default as an ExecuTorch backend. Since I use a Snapdragon 8 Gen 2 with Armv9-A architecture on Android, it is a good fit as all operations listed in Section 3.1 needed to execute the ResNet50 v1.5 model are supported.

4.2.2. Vulkan

Vulkan [52] is an open standard for graphics programming developed by the Khronos Group. It is known as a successor to OpenGL by the community, allowing for more performance through hardware-aware programming. Features such as manual memory management and support for multiple processors allow for very effective load balancing in exchange for high initialization overhead [7], [56]. As a graphics language, it is designed for highly parallel efficient GPU tasks and can compute ML workloads by splitting them into multiple parallel subtasks. The Adreno 740 GPU supporting Vulkan 1.3 is a good fit for this backend as it supports the latest standard and is already integrated into ExecuTorch.

4.2.3. Qualcomm AI Engine Direct

The Qualcomm AI Engine Direct is a Software Development Kit (SDK) that provides a low-level, unified API for ML development. It is designed to improve the performance of ML models on Qualcomm's AI accelerators on the CPU, GPU, and HTP. Qualcomm AI Engine Direct is designed as a hardware abstraction API for clean software separation to different hardware cores. The SDK allows developers to manage the trade-off between core-specific library capabilities and memory usage, ensuring minimal memory usage while delivering the highest performance [46]. With this backend, I can use the dedicated NPU of the chip and get a significant performance boost, which is discussed in Section 4.5. The HTP is currently the only component supported by this backend, but this is good enough for my purposes as other backends like XNNPACK or Vulkan support the remaining hardware.

[2] GitHub repository: https://github.com/google/XNNPACK

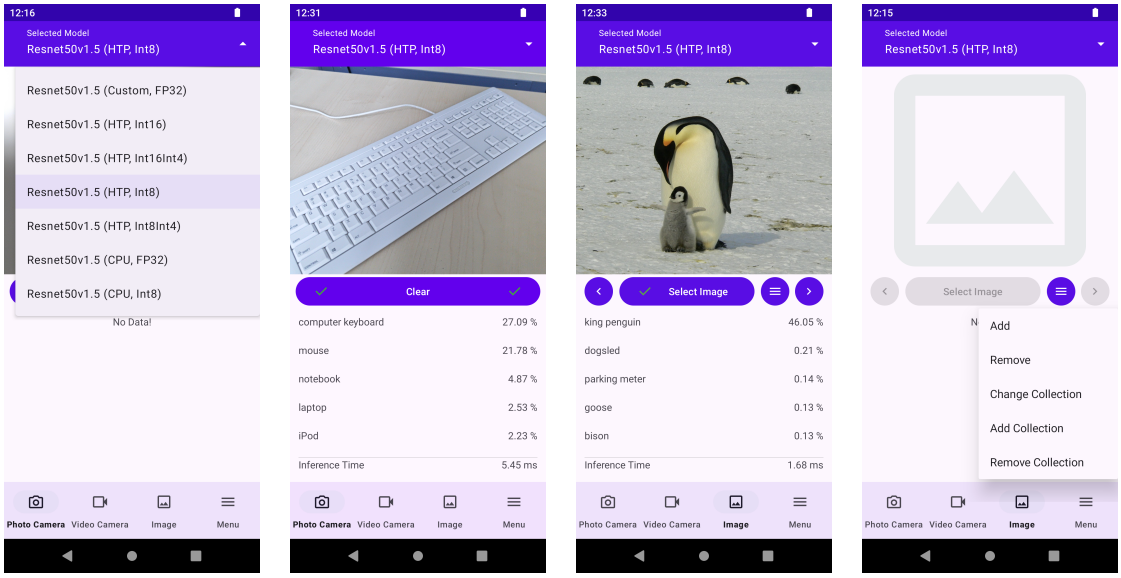
4.3. Custom Operator

Custom operators are operators defined by the user of the ExecuTorch platform as needed and used to extend or overwrite parts of the ATen library. ExecuTorch provides two methods to register a custom operator in both the ExecuTorch runtime and PyTorch. The first is based on macros provided by ExecuTorch and PyTorch. It allows fast development since only a single file needs to be changed to adapt the custom operator. However, it does not allow selective build as it is always registered in the ExecuTorch runtime. The second method is more complex, requiring an additional registration file where the operator's declaration is available. It has the advantage of supporting selective build and a more centralized management of custom operators. The declaration is then parsed, and additional code is generated to obtain a registration library, which acts as an interface to the ExecuTorch platform. The custom operator source code is then compiled into a kernel library and linked to the registration library. Finally, the registration library can be linked to the desired application or shared library, which executes the lowered models.

4.4. Android App

To run the lowered models on the Snapdragon 8 Gen 2 SoC, I implemented an interface via an Android app that is extensible for other models to infer vision data. The app includes key features, such as the ability to perform inference on real-time input from the camera and inference from the storage of the used device through images.

Figure 4.1a shows the available Snapdragon 8 Gen 2 SoC models. The selection menu adapts to the device by comparing the available hardware to the available lowered models. So, a mobile device without the HTP would only see the models that can run on the CPU. Figure 4.1b shows the interface for photographing the environment with the device camera. When a photo is taken, the resulting image is automatically evaluated by the selected model, and the top five guesses are returned, along with the model's inference time. The details shown for the image can be expanded to show additional data that other models might produce. The "Video Camera" interface, selectable in the bottom navigation bar, is very similar to the "Photo Camera" interface. It continuously takes photos of the environment and processes them immediately, providing real-time feedback to the observation. The details also show the possible frames per second that can be processed by the model, which can help to evaluate if the selected model is a good fit for the current task.



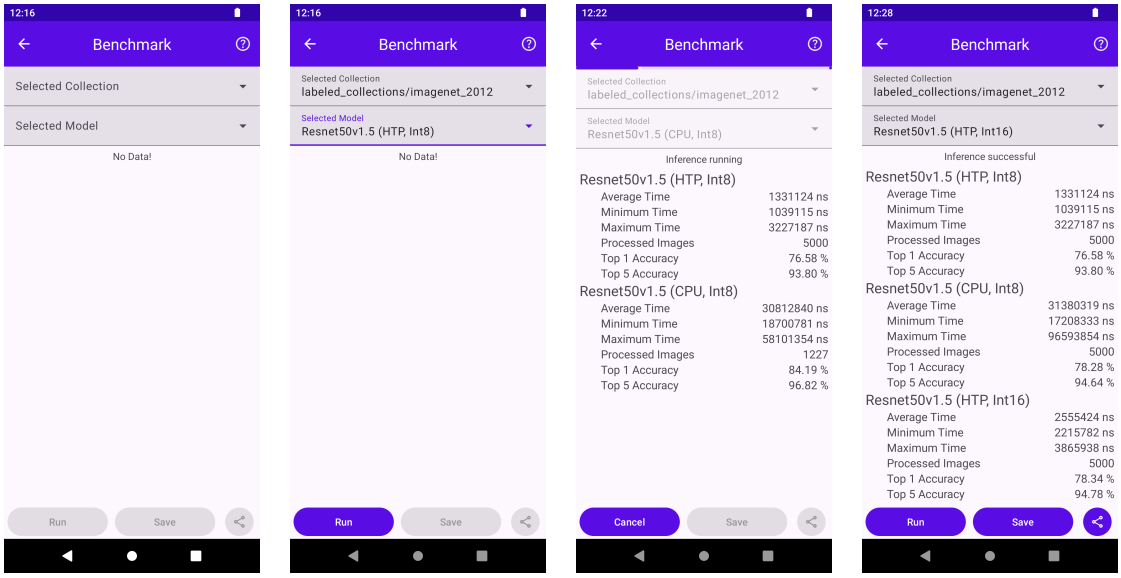
(a) Model Selection (b) Photo Camera (c) Image from Storage (d) Image Menu

Figure 4.1. The user interface for selecting and running a model on real-time input and stored input.

The other interface shown in Figure 4.1c loads images from the device's storage and evaluates them immediately after selection by the current model. It shows the same details as the "Photo Camera" interface. The left and right arrows select the previous or next image in the current collection. In addition, one can use the "Select Image" button to choose a specific image from the collection. A collection contains several images and can be used to perform benchmarks. With the menu shown in Figure 4.1d, one can add and remove images from the current collection or change the collection. Furthermore, the option to create and remove a collection is provided to categorize the images better.

The Android app also offers a benchmark suite for the models, located in the "Menu" option of the bottom navigation bar. The benchmarks include a warm-up phase of passing 25 images through the model. 25 has proven to be a good value for achieving benchmarks without a high initial runtime.

Figure 4.2a shows the default setup the user will see when the benchmark suite starts. First, one has to select a collection, which can be a predefined collection with labels or a user-defined collection instanced in the image interface. Note that only a collection with labels can show the model's accuracy, as the actual label is needed for the calculation. Next, a user can select a model on which to run the benchmark. The results of multiple models are displayed based on the collection, i.e., the model results are bound to the collection. In Figure 4.2c, a progress bar at the top shows the number of images processed. In addition, the model results are updated in real-time, helping the user to catch early errors. When the benchmark



(a) Empty (b) Prepared (c) In Progress (d) Finished

Figure 4.2. The benchmark suite of my Android app to test performance across multiple datasets and models.

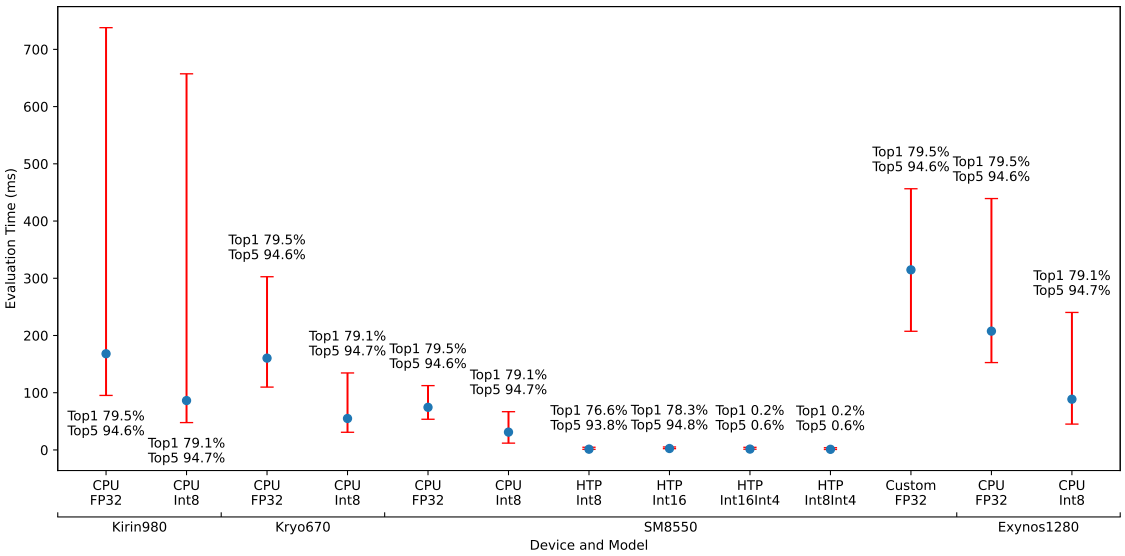
is finished, shown in Figure 4.2d, the model results can be saved to local storage or shared with other applications for transfer or post-processing. The model results are saved in a JSON format as it is easy to process and human-readable.

4.5. Performance

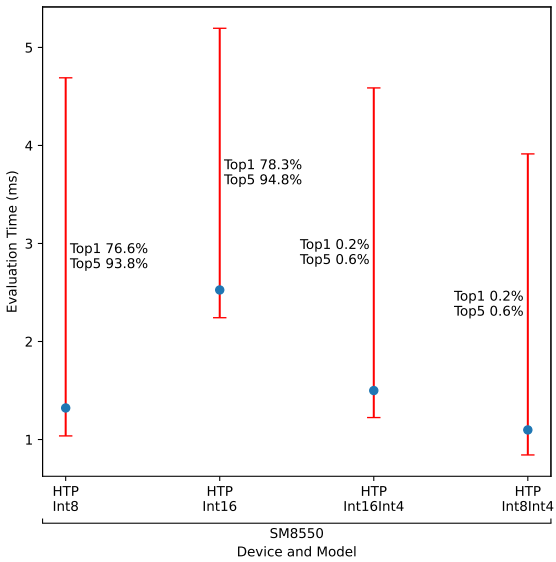
This section compares the performance of the ResNet50 v1.5 model benchmarked with my Android app, introduced in Section 4.4. First, we evaluate different hardware and quantization schemes for the ResNet50 v1.5. Lastly, we analyze the efficiency of the HTP lowered model based on the theoretical peak performance. At the time of writing, the Vulkan backend for the GPU did not work. Therefore, it is missing in the benchmark results.

4.5.1. Hardware & Quantization

The Android app contains a benchmark dataset of 5000 random images from the ImageNet validation dataset. I used this to generate the benchmark results in Figure 4.3. The blue dot shows the average runtime and the red bar shows the range based on the minimum and maximum runtime collected during the benchmark. The benchmark consists of the ResNet50 v1.5 model lowered to multiple quantization and different backends, such as XNNPACK for CPU and Qualcomm AI Engine Direct for HTP. I used Post Training Quantization (PTQ) for the quantized models shown



(a) Benchmark on all hardware.



(b) Benchmark only on HTP.

Figure 4.3. The ResNet50 v1.5 model for the benchmark was lowered to different hardware and data types and evaluated with 5000 random ImageNet validation samples. Figure 4.3a shows the results on different devices and hardware. Figure 4.3b shows the same results as Figure 4.3a but visualizes only the HTP models for better clarity. The blue dot shows the average runtime and the red bar shows the range based on the minimum and maximum runtime collected during the benchmark.

in Figure 4.3. PTQ was trained with 1500 ImageNet validation samples, excluding the samples used by the benchmark. The FP32 benchmark serves as a baseline for accuracy as I only used a subset of the ImageNet validation dataset and did not achieve the original 80.86 % accuracy achieved by the original ResNet50 v1.5 model [53].

For the CPU, I used FP32 to represent the standard version and Int8 for a quantized version. Throughout the benchmark, we see that Int8 quantization halves or almost halves the time needed to run the model while maintaining almost FP32 accuracy. We also see that the accuracy is maintained across multiple devices, indicating that XNNPACK is a portable library across different hardware.

The HTP models gain significantly in performance as they run on special matrix-matrix multiplication hardware. We see that the accuracy of the HTP Int8 model does not match the other CPU Int8 models. The change in accuracy may be due to numerical issues based on different orders of the underlying mathematical operations. In Figure 4.3b, the Int8 model cuts the runtime in half compared to the Int16 model. I also tried the Int4 quantization of the weights notated by Int16Int4 or Int8Int4, where the first is the activation data type and the second is the weight data type. The model runs "successfully" as it can be executed, but we see an accuracy of 0.2 %. The extremely low accuracy could be attributed to the Int4 weights, which need more precision to extract the correct class. Alternatively, the quantization process through ExecuTorch's Python interface may not work correctly, as PyTorch, which ExecuTorch uses, does not natively support Int4 types. Another explanation could be a problem in the AI Engine direct backend with Int4 weights or another issue related to Int4 weights.

The SM8550 devices also show the performance of my custom implementation, which runs on the CPU. My custom implementation is about three times slower than the CPU model that used XNNPACK as backend. The significant performance difference could be attributed to the backend itself. For the convolutions, I use Libxsmm, which does not natively support Android, unlike XNNPACK. Also, XNNPACK might have a specific implementation for small convolutions, where I took a more general approach for all convolutions. Additionally, ExecuTorch's lowering process could be the cause of ResNet50 v1.5's performance improvement or other unknown reasons.

4.5.2. Theoretical HTP Performance

To better understand the implementation's efficiency, we compare the ResNet50 v1.5 models achieved performance with an approximated peak performance on the HTP. Therefore, we first approximate the operation needed to classify an input on a ResNet50 v1.5 model and then use the theoretical operations of 18.8 TOPS with Int4 weights stated in Section 2.2. We will calculate the TOPS for the Int8, and Int16 model based on the size of the datatype i.e. $\text{TOPS}_{\text{IntX}} = 18.8\,\text{TOPS} \cdot \frac{4\,\text{bits}}{X\,\text{bits}}$.

To approximate the number of operations required for a convolution, we use the filter and output dimensions since each filter application results in a single output pixel. We use the notation introduced in Section 3 for the approximation formulas. Thus, we derive $\text{ops}_{\text{Conv}} = 2 \cdot (R \cdot S \cdot C) \cdot (H' \cdot W') \cdot K$. The part $(R \cdot S \cdot C)$ represents the data on which the filter operates, and we perform two operations, multiplication and addition, on each of the filter weights. Furthermore, the process results in a $(H' \cdot W')$ large output, which is repeated for each filter, i.e., the output dimension K. The striding of the convolution is automatically included as we work on the dimensions of the output.

Approximating the post-processing, i.e., applying batch normalization, ReLU, and identity shortcut, is relatively easy. Batch normalization has the formula $y = \frac{x - E[X]}{\sqrt{Var[x] + \epsilon}} \cdot \gamma + \beta$, so we count six operations, resulting in $\text{ops}_{Batch} = 6 \cdot (H' \cdot W' \cdot K)$. The ReLU function performs one operation, giving the formula $\text{ops}_{ReLU} = (H' \cdot W' \cdot K)$, and an identity shortcut performs one addition operation, giving the same formula $\text{ops}_{Id} = (H' \cdot W' \cdot K)$. Note that a projection shortcut is represented by the combination: $\text{ops}_{\text{Conv}} + \text{ops}_{Batch} + \text{ops}_{Id}$.

The max pooling can be described by a formula similar to that of the convolutions: $\text{ops}_{\text{MaxPool}} = (C \cdot 3 \cdot 3) \cdot (H' \cdot W' \cdot C)$. Getting the maximum value is only one operation, and the max pooling is fixed to a size of 3×3. In addition, the channel dimension remains the same.

The global average pool is described by $\text{ops}_{\text{GAvgPool}} = (H \cdot W \cdot C) + C$. The summation of all elements is represented by $(H \cdot W \cdot C)$, and the averaging is a single operation on each channel.

Finally, the fully connected layer does $\text{ops}_{\text{FullyC}} = 2 \cdot (H \cdot W)$ operations since each column of the weight matrix is used once for multiplication and addition.

We then use these formulas to construct the ResNet50 v1.5 model and sum up each operation, resulting in 9 299 929 600 operations. For a more detailed explanation of how the operations were calculated, see Appendix A.

The TOPS of the ResNet50 v1.5 model are then calculated based on the benchmark. The benchmark result for Int8 is 1 322 008 ns, and for Int16 is 2 525 768 ns. So we calculate for Int8 $\frac{9299929600 \cdot 10^{-12}}{1322008 \cdot 10^{-9}} \approx 7.0347\,\text{TOPS}$ and for Int16 $\frac{9299929600 \cdot 10^{-12}}{2525768 \cdot 10^{-9}} \approx 3.6820\,\text{TOPS}$. Then we scale the theoretical peak performance to the correct data type, resulting in $18.8\,\text{TOPS} \cdot \frac{4\,\text{bits}}{8\,\text{bits}} = 9.4\,\text{TOPS}$ for Int8 and $18.8\,\text{TOPS} \cdot \frac{4\,\text{bits}}{16\,\text{bits}} = 4.7\,\text{TOPS}$ for Int16. Further a comparison results in $\frac{7.0347\,\text{TOPS}}{9.4\,\text{TOPS}} \approx 74.8\,\%$ for Int8 and $\frac{3.6820\,\text{TOPS}}{4.7\,\text{TOPS}} \approx 78.3\,\%$ for Int16 of theoretical performance.

Achieving 74 % to 78 % of theoretical peak performance is a good result because not every component of the ResNet50 v1.5 model can be described as a MatMul and thus results in lower performance. For example, max pooling uses maximum operation, and global average pooling uses summation reduction. In addition, batch normalization may also degrade performance because it requires square roots and divisions to be calculated. The HTP lowered model achieves good performance, as it is nearly impossible to achieve full theoretical performance. Note that we may see improved performance as ExecuTorch development progresses, as it is still in the alpha stage.

5. RELATED WORK

Matrix-matrix multiplication works for most of the operations required to compute a CNN. In addition, using the HTP allows us to gain a significant performance boost, but it is limited to a matrix-matrix multiplication-like implementation and similar operations. As more complex models emerge, such as the YOLOv10 [57], only some parts of the model can benefit from the HTP or similar coprocessors.

A solution to this problem is the Xilinx/AMD "Versal AI Engine", which consists of an array of up to 400 AI Engine tiles with manageable data paths, pictured in Figure 5.1. The data paths ensure high flexibility for different tasks [4], [33] since the implementation also manages data streams between tiles. The flexibility allows it to be more future-proof for newer, more complex models [1], as an adaption to complex operations can be performed by data stream manipulation. In addition, the complete Xilinx Versal SoC also includes adaptive hardware with a high-speed connection to the AI Engines through the integrated network-on-chip. The adaptive hardware allows the SoC to adapt to any scenario or computational logic a model may require. Note that the Xilinx Versal SoC is considerably larger in chip size than the Snapdragon 8 Gen 2 SoC, as it is not designed for mobile applications. Nevertheless, the architecture of AI Engines is used in the latest mainstream SoCs, known as AMD Ryzen AI [3]. Ryzen AI includes a NPU based on the AMD XDNA architecture [2] derived from the Versal AIEngines architecture.

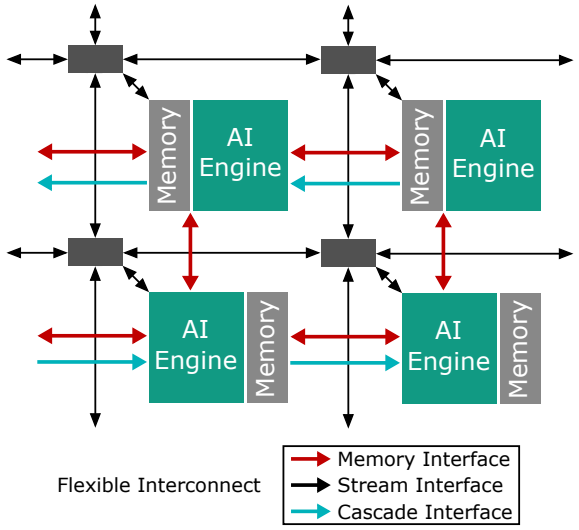


Figure 5.1. AI Engine architecture with dedicated connectivity between tiles, each holding its data memory [4].

Early exiting is a strategy that holds great promise in saving computational resources and inference time. It was first introduced by Bolukbasi et al. [12], who proposed two schemes. The first scheme introduces a policy between layers that decides whether the model's prediction is good enough to be returned as a reasonable prediction. If not, the model proceeds to the next layer, checks the corresponding policy, and repeats these steps until the end of the whole model. The second proposed scheme was a network selection approach, which means that the models are sorted by complexity in an acyclic graph, starting with lower complexity. Each model node is divided by a policy node that determines the confidence of the parent model node and jumps to a more complex model to increase accuracy if needed. Otherwise, the prediction is returned as is. A network selection of AlexNet → GoogleLeNet → ResNet50 achieved a speedup of 2.8 while maintaining the accuracy of ResNet50 within 1 %. An overview of other early exit strategies is given in [50].

An adaptation for early exiting on Large Language Model (LLM) has been proposed by [55], as the task of natural language processing has become more popular. A larger model generally correlates with improved performance, generality, and high computational cost, which can only be covered by robust server infrastructure. Early exiting significantly reduced computational resources, allowing the model to be processed in a local environment, ensuring the privacy of the input and less cost for server infrastructure in response to user demand.

Xin et al. [61] have proposed another adaptation for early exiting the BERT model. Through early exiting, the BERT model can save up to 40 % of the inference time. This significant reduction in inference time enables real-time application and inference on edge devices like mobile phones.

Facebook (now Meta) has made great efforts to deploy Deep Neural Networks (DNNs) on edge devices, mainly mobile phones [60]. They provide a comprehensive overview of the hardware used by billions of users and the techniques used to deploy ML models on these devices. The challenges of serving these many devices, such as high variability in available performance, different coprocessors and accelerators, model size, and connectivity to access model weights, require a versatile framework. ExecuTorch addresses most of these issues through its adaptable lowering process and multiple backends, as described in Section 4.

6. CONCLUSION AND FUTURE WORK

With the rise of DL applications, the demand for fast inference increases. In addition, the number of parameters a model uses increases to fit more general tasks. A SoC can satisfy these requirements, which often contain specialized hardware to accelerate the computation of MatMuls. In this thesis, we discussed a high-performance implementation for a CNN by a blocked implementation of the convolutions, max pooling, and global average pooling. Furthermore, we looked at the ExecuTorch platform, an end-to-end solution for running model inference on edge devices, using multiple backends to target specific hardware on the SoC. ResNet50 v1.5 was chosen as an example DL model, and we discussed a comprehensive performance overview by comparing my custom implementation to an existing one. In addition, I evaluated different ExecuTorch backends on various devices and hardware and presented an extensible Android app for inference and benchmarking on mobile devices.

As demonstrated in the evaluation, my custom implementation of the ResNet50 v1.5 model can compete with existing implementations and achieve up to 1.85 times faster performance with my multicore "ConvBlock"[3] implementation. Using the Qualcomm AI Engine Direct backend of ExecuTorch, which enabled the use of the HTP, resulted in a significant speedup compared to the CPU. The runtime of the Int8 model on HTP compared to the Int8 model on CPU resulted in a speedup of 23.6, as the HTP has dedicated hardware for Matrix-Matrix multiplication. Therefore, dedicated hardware similar to the HTP should be used to achieve the best efficiency.

Future work could include early exiting, as described in Section 5, to further reduce inference time and memory usage. The use of early exiting could result in a significant reduction in on-device inference time while maintaining the same performance. Furthermore, one could take advantage of more features of the ExecuTorch platform. For example, one could write a custom backend for specific hardware to enable model inference on additional hardware. Also, one could explore the selective build feature and how to automatically select the correct operations required by different models, leading to reduced binary file size.

[3] "ConvBlock" combines the convolution, batch normalization, and ReLU operation.

REFERENCE

[1] G. Alok. "Architecture ApocalypseDream Architecture for Deep Learning Inference andCompute VERSAL AI Core." [access date 2024-08-09], Xilinx Inc. (Aug. 9, 2024), [Online]. Available: https://download.amd.com/docnav/documents/aem/white_papers-EW2020-Deep-Learning-Inference-AICore.pdf.

[2] AMD. "AMD XDNA™ Architektur." [access date 2024-09-01], Advanced Micro Devices, Inc. (Sep. 1, 2024), [Online]. Available: https://www.amd.com/en/technologies/xdna.html.

[3] AMD. "The Future of AI PCs Gets Even Better with AMD." [access date 2024-09-01], Advanced Micro Devices, Inc. (Sep. 1, 2024), [Online]. Available: https://www.amd.com/en/products/processors/consumer/ryzen-ai.html.

[4] AMD and Xilinx. "AI Engine: Meeting the Compute Demands of Next-Generation Applications." [access date 2024-08-23], Advanced Micro Devices, Inc. (Aug. 23, 2024), [Online]. Available: https://www.xilinx.com/products/technology/ai-engine.html.

[5] Apple. "Apple introduces M4 chip." [access date 2024-08-10], Apple Inc. (Aug. 10, 2024), [Online]. Available: https://www.apple.com/newsroom/2024/05/apple-introduces-m4-chip/.

[6] Arm Developer. "Arm Architecture Reference Manual for A-profile architecture." [access date 2024-08-13], Arm Limited. (Aug. 13, 2024), [Online]. Available: https://developer.arm.com/documentation/ddi0487/ka/?lang=en.

[7] Arm Developer. "Comparison of Vulkan API vs OpenGL ES API on Arm." [access date 2024-08-13]. (Aug. 13, 2024), [Online]. Available: https://developer.arm.com/Additional%20Resources/Video%20Tutorials/Comparison%20of%20Vulkan%20API%20vs%20OpenGL%20ES%20API%20on%20Arm.

[8] Arm Developer. "Development of the Arm architecture." [access date 2024-08-13], Arm Limited. (Aug. 13, 2024), [Online]. Available: https://developer.arm.com/documentation/102404/0201/Development-of-the-Arm-architecture?lang=en.

[9] B. Arslan, S. Gunduz, and S. Sagiroglu, "A review on mobile threats and machine learning based detection approaches," in *2016 4th International Symposium on Digital Forensic and Security (ISDFS)*, IEEE, Apr. 2016, pp. 7–13. DOI: 10.1109/ISDFS.2016.7473509.

[10] A. Baldominos, A. Cervantes, Y. Saez, and P. Isasi, "A Comparison of Machine Learning and Deep Learning Techniques for Activity Recognition using Mobile Devices," *Sensors*, vol. 19, no. 3, p. 521, Jan. 2019. DOI: 10.3390/s19030521. [Online]. Available: https://www.mdpi.com/1424-8220/19/3/521.

[11] J. Bergstra *et al.*, "Theano: Deep learning on gpus with python," in *NIPS 2011, BigLearning Workshop, Granada, Spain*, Citeseer, vol. 3, 2011.

[12] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive Neural Networks for Efficient Inference," *Proceedings of the 34th International Conference on Machine Learning, PMLR 70:527-536, 2017*, Feb. 2017. DOI: 10.48550/ARXIV.1702.07811. arXiv: 1702.07811 [cs.LG].

[13] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, May 2011. DOI: 10.1145/1941487.1941507.

[14] A. Breuer and S. Remke. "Hello SME." [access date 2024-08-10], Friedrich Schiller University Jena. (Aug. 10, 2024), [Online]. Available: https://scalable.uni-jena.de/opt/sme/.

[15] A. Cabrera, S. Hitefield, J. Kim, S. Lee, N. R. Miniskar, and J. S. Vetter, "Toward Performance Portable Programming for Heterogeneous Systems on a Chip: A Case Study with Qualcomm Snapdragon SoC," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, Sep. 2021, pp. 1–7. DOI: 10.1109/HPEC49654.2021.9622794.

[16] S. Chatterjee and P. Zielinski, "On the Generalization Mystery in Deep Learning," Mar. 2022. DOI: 10.48550/ARXIV.2203.10036. arXiv: 2203.10036 [cs.LG].

[17] clamchowder. "Qualcomm's Hexagon DSP, and now, NPU." [access date 2024-08-10]. (Oct. 4, 2023), [Online]. Available: https://chipsandcheese.com/2023/10/04/qualcomms-hexagon-dsp-and-now-npu/.

[18] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998. DOI: 10.1109/99.660313.

[19] M. Dukhan, "The Indirect Convolution Algorithm," Jul. 2019. DOI: 10.48550/ARXIV.1907.02129. arXiv: 1907.02129 [cs.CV].

[20] ExecuTorch. "Welcome to the ExecuTorch Documentation." [access date 2024-08-12], The Linux Foundation. (Aug. 12, 2024), [Online]. Available: https://pytorch.org/executorch/0.3/.

[21] M. E. Fathy, V. M. Patel, and R. Chellappa, "Face-based Active Authentication on mobile devices," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, Apr. 2015, pp. 1687–1691. DOI: 10.1109/ICASSP.2015.7178258.

[22] E. Georganas *et al.*, "High-Performance Deep Learning via a Single Building Block," Jun. 2019. DOI: 10.48550/ARXIV.1906.06440. arXiv: 1906.06440 [cs.LG].

[23] E. Georganas *et al.*, "Tensor Processing Primitives: A Programming Abstraction for Efficiency and Portability in Deep Learning and HPC Workloads," SC '21, Apr. 12, 2021. DOI: 10.1145/3458817.3476206. arXiv: 2104.05755 [cs.AI].

[24] GitHub. "The world's most widely adopted AI developer tool." [access date 2024-08-08], GitHub, Inc. (Aug. 8, 2024), [Online]. Available: https://github.com/features/copilot.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," Dec. 2015. DOI: 10.48550/ARXIV.1512.03385. arXiv: 1512.03385 [cs.CV].

[26] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, Nov. 2016, pp. 981–991. DOI: 10.1109/sc.2016.83.

[27] hollance. "The Neural Engine — what do we know about it?" [access date 2024-08-28]. (Aug. 28, 2024), [Online]. Available: https://github.com/hollance/neural-engine.

[28] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015. DOI: 10.48550/ARXIV.1502.03167.

[29] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[30] L. Jones. "Microsoft's GitHub Copilot Faces Financial Challenges despite Popularity." [access date 2024-08-08]. (Aug. 8, 2024), [Online]. Available: https://winbuzzer.com/2023/10/10/microsofts-github-copilot-faces-financial-challenges-despite-popularity-xcxwbn/.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017. DOI: 10.1145/3065386.

[32] A. Lavin and S. Gray, "Fast Algorithms for Convolutional Neural Networks," Sep. 2015. DOI: 10.48550/ARXIV.1509.09308. arXiv: 1509.09308 [cs.NE].

[33] J. Lei, J. Flich, and E. S. Quintana-Ortí, "Toward Matrix Multiplication for Deep Learning Inference on the Xilinx Versal," in *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Mar. 2023, pp. 227–234. DOI: 10.1109/PDP59025.2023.00043.

[34] M. Lin, Q. Chen, and S. Yan, "Network In Network," Dec. 2013. DOI: 10.48550/ARXIV.1312.4400. arXiv: 1312.4400 [cs.NE].

[35] J. Luke, R. Joseph, and M. Balaji, "Impact of image size on accuracy and generalization of convolutional neural networks," *Int. J. Res. Anal. Rev.(IJRAR)*, vol. 6, no. 1, pp. 70–80, 2019.

[36] E. Mahurin. "Qualcomm® Hexagon™ NPU." [access date 2024-08-28], Qualcomm Technologies, Inc. (Aug. 29, 2023), [Online]. Available: https://hc2023.hotchips.org/assets/program/conference/day2/ML%20Inference/HC2023%20Qualcomm%20Hexagon%20NPU.pdf.

[37] O. Mortensen. "How Many Users Does ChatGPT Have? Statistics and Facts (2024)." [access date 2024-08-08]. (Aug. 8, 2024), [Online]. Available: https://seo.ai/blog/how-many-users-does-chatgpt-have.

[38] NVIDIA. "ResNet v1.5 for PyTorch." [access date 2024-08-05], NVIDIA Corporation. (Aug. 5, 2024), [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/resources/resnet_50_v1_5_for_pytorch.

[39] OnePlus. "OnePlus 12R." [access date 2024-08-10], Shenzhen OnePlus Science & Technology Co., Ltd. (Aug. 10, 2024), [Online]. Available: https://www.oneplus.com/us/12r/specs.

[40] OpenAI. "Hello GPT-4o." [access date 2024-08-08], OpenAI, Inc. (Aug. 8, 2024), [Online]. Available: https://openai.com/index/hello-gpt-4o/.

[41] OpenAI. "Pricing." [access date 2024-08-08], OpenAI, Inc. (Aug. 8, 2024), [Online]. Available: https://openai.com/api/pricing/.

[42] R. Poya, A. J. Gil, and R. Ortigosa, "A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics," *Computer Physics Communications*, vol. 216, pp. 35–52, Jul. 2017. DOI: 10.1016/j.cpc.2017.02.016.

[43] PyTorch. "Definition of the Core ATen Operator Set." [access date 2024-08-29], The Linux Foundation. (Aug. 29, 2024), [Online]. Available: https://pytorch.org/executorch/stable/ir-ops-set-definition.html.

[44] PyTorch. "Export IR Specification." [access date 2024-08-29]. (Aug. 29, 2024), [Online]. Available: https://pytorch.org/executorch/stable/ir-exir.html#edge-dialect.

[45] PyTorch. "PyTorch C++ API." [access date 2024-08-19], The Linux Foundation. (Aug. 19, 2024), [Online]. Available: https://pytorch.org/cppdocs/.

[46] Qualcomm. "Qualcomm AI Engine Direct SDK." [access date 2024-08-13], Qualcomm Technologies, Inc. (Aug. 13, 2024), [Online]. Available: https://www.qualcomm.com/developer/software/qualcomm-ai-engine-direct-sdk#getstarted.

[47] Qualcomm. "Snapdragon 8 Gen 2 Mobile Platform." [access date 2024-08-05], Qualcomm Technologies, Inc. (Aug. 5, 2024), [Online]. Available: https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-8-gen-2-mobile-platform.

[48] Qualcomm. "Snapdragon 8 Gen 2 Product Brief." [access date 2024-08-10], f Qualcomm Technologies International, Ltd. (Jul. 5, 2023), [Online]. Available: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/Snapdragon-8-Gen-2-Product-Brief.pdf.

[49] Qualcomm developer network. "AI hardware cores/accelerators." [access date 2024-08-08], Qualcomm Technologies, Inc. (Aug. 8, 2024), [Online]. Available: https://developer.qualcomm.com/hardware/qualcomm-innovators-development-kit/ai-resources-overview/ai-hardware-cores-accelerators.

[50] S. Scardapane, M. Scarpiniti, E. Baccarelli, and A. Uncini, "Why should we add early exits to neural networks?" *Cognitive Computation, 2020*, vol. 12, no. 5, pp. 954–966, Apr. 27, 2020. DOI: 10.1007/s12559-020-09734-4. arXiv: 2004.12814 [cs.NE].

[51] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," Sep. 2014. DOI: 10.48550/ARXIV.1409.1556. arXiv: 1409.1556 [cs.CV].

[52] The Khronos® Group Inc. "Vulkan." [access date 2024-08-31]. (Aug. 31, 2024), [Online]. Available: https://www.vulkan.org.

[53] Torch Contributors. "Resnet50." [access date 2024-08-05], The Linux Foundation. (Aug. 5, 2024), [Online]. Available: https://pytorch.org/vision/main/models/generated/torchvision.models.resnet50.html.

[54] R. Triggs. "Snapdragon 8 Gen 2 deep dive: Everything you need to know." [access date 2024-08-10]. (Feb. 8, 2024), [Online]. Available: https://www.androidauthority.com/snapdragon-8-gen-2-explained-3231826/.

[55] F. Valade, "Accelerating Large Language Model Inference with Self-Supervised Early Exits," Jul. 2024. DOI: 10.48550/ARXIV.2407.21082. arXiv: 2407.21082 [cs.CL].

[56] Vulkan Documentation. "Vulkan Specification Fundamentals." [access date 2024-08-13], The Khronos Group Inc. (Aug. 13, 2024), [Online]. Available: https://docs.vulkan.org/spec/latest/chapters/fundamentals.html.

[57] A. Wang *et al.*, "YOLOv10: Real-Time End-to-End Object Detection," May 2024. DOI: 10.48550/ARXIV.2405.14458. arXiv: 2405.14458 [cs.CV].

[58] D. Weir, S. Rogers, R. Murray-Smith, and M. Löchtefeld, "A user-specific machine learning approach for improving touch accuracy on mobile devices," in *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '12, Cambridge, Massachusetts, USA: Association for Computing Machinery, Oct. 2012, pp. 465–476. DOI: 10.1145/2380116.2380175. [Online]. Available: https://doi.org/10.1145/2380116.2380175.

[59] Wikipedia. "System on a chip." [access date 2024-08-08], Wikimedia Foundation, Inc. (Aug. 8, 2024), [Online]. Available: https://en.wikipedia.org/wiki/System_on_a_chip.

[60] C.-J. Wu *et al.*, "Machine Learning at Facebook: Understanding Inference at the Edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, Feb. 2019, pp. 331–344. DOI: 10.1109/HPCA.2019.00048.

[61] J. Xin, R. Tang, J. Lee, Y. Yu, and J. Lin, "DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference," Apr. 2020. DOI: 10.48550/ARXIV.2004.12993. arXiv: 2004.12993 [cs.CL].

[62] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A First Look at Deep Learning Apps on Smartphones," Nov. 2018. DOI: 10.48550/ARXIV.1812.05448. arXiv: 1812.05448 [cs.LG].

LIST OF FIGURES

2.1 ResNet50 v1.5 architecture 12
2.2 Input size benchmark 14
2.3 Hexagon Processor overview 15

3.1 Blocking of a convolution 17
3.2 Comparison Custom vs ATen 22

4.1 Android app UI for inference 28
4.2 Android app UI for benchmarks 29
4.3 Lowered model performance 30

5.1 Xilinx Versal AI Engine Array 34

LIST OF TABLES

2.1 HTP data types compositions 15

APPENDIX A

RESNET50 V1.5 - NUMBER OF OPERATIONS

This section shows the complete calculation of the theoretical performance of the ResNet50 v1.5 model. We use the basic formulas derived in Section 4.5.2:

$$\begin{aligned}
\text{Convolution:} &\quad \text{ops}_{\text{Conv}} = 2 \cdot (R \cdot S \cdot C) \cdot (H' \cdot W') \cdot K \\
\text{Batch Normalization:} &\quad \text{ops}_{Batch} = 6 \cdot (H' \cdot W' \cdot K) \\
\text{ReLU:} &\quad \text{ops}_{ReLU} = (H' \cdot W' \cdot K) \\
\text{Identity Shortcut:} &\quad \text{ops}_{Id} = (H' \cdot W' \cdot K) \\
\text{Projection Shortcut:} &\quad \text{ops}_{\text{Conv}} + \text{ops}_{Batch} + \text{ops}_{Id} \\
\text{Max Pool:} &\quad \text{ops}_{\text{MaxPool}} = (3 \cdot 3) \cdot (H' \cdot W' \cdot C) \\
\text{Global Average Pool:} &\quad \text{ops}_{\text{GAvgPool}} = (H \cdot W \cdot C) + C \\
\text{Fully Connected Layer:} &\quad \text{ops}_{\text{FullyC}} = 2 \cdot (H \cdot W)
\end{aligned}$$

We combine these to represent a bottleneck block with an identity or projection shortcut. The width, height, and input channel dimensions are retrieved based on the previous addition in the sum. For the identity shortcut, we get the following:

$$\begin{aligned}
\text{ops}^{K}_{\text{BlockId}} = {} & \text{ops}^{1\times1,K}_{\text{Conv}} + \text{ops}^{K}_{Batch} + \text{ops}^{K}_{ReLU} \\
& + \text{ops}^{3\times3,K}_{\text{Conv}} + \text{ops}^{K}_{Batch} + \text{ops}^{K}_{ReLU} \\
& + \text{ops}^{1\times1,4\cdot K}_{\text{Conv}} + \text{ops}^{4\cdot K}_{Batch} + \text{ops}^{4\cdot K}_{ReLU} \\
& + \text{ops}^{4\cdot K}_{Id}
\end{aligned}$$

For the projection shortcut, we get the following:

$$\begin{aligned}
\text{ops}^{K,\text{stride}}_{\text{BlockPj}} = {} & \text{ops}^{1\times1,K}_{\text{Conv}} + \text{ops}^{K}_{Batch} + \text{ops}^{K}_{ReLU} \\
& + \text{ops}^{3\times3,K}_{\text{Conv}} + \text{ops}^{K}_{Batch} + \text{ops}^{K}_{ReLU} \\
& + \text{ops}^{1\times1,4\cdot K,\text{stride}}_{\text{Conv}} + \text{ops}^{4\cdot K}_{Batch} + \text{ops}^{4\cdot K}_{ReLU} \\
& + \text{ops}^{1\times1,4\cdot K}_{\text{Conv}^0} + \text{ops}^{4\cdot K}_{Batch} + \text{ops}^{4\cdot K}_{Id}
\end{aligned}$$

Note that $\text{ops}^{1\times1,4\cdot K}_{\text{Conv}^0}$ is named with Conv^0, so the height, width, and input channel dimension refer to the input of the first convolution in the summation.

Applying these formulas to the complete ResNet50 v1.5 architecture with input dimensions of $3 \times 224 \times 224$, as discussed in Section 2.1, yields the following overview. The $\rightarrowtail$ indicates the calculated number of operations used by this specific operation.

$$
\begin{aligned}
\text{ops} = \text{ops}^{7\times7,64,\text{stride=2}}_{\text{Conv}} &\rightarrowtail 236027904 \\
+ \text{ops}^{64}_{\textit{Batch}} &\rightarrowtail 4816896 \\
+ \text{ops}^{64}_{\textit{ReLU}} &\rightarrowtail 802816 \\
+ \text{ops}_{\text{MaxPool}} &\rightarrowtail 115605504 \\
\\
+ \text{ops}^{64,\text{stride=1}}_{\text{BlockPj}} &\rightarrowtail 784752640 \\
+ \text{ops}^{64}_{\text{BlockId}} &\rightarrowtail 445964288 \\
+ \text{ops}^{64}_{\text{BlockId}} &\rightarrowtail 445964288 \\
\\
+ \text{ops}^{128,\text{stride=2}}_{\text{BlockPj}} &\rightarrowtail 959666176 \\
+ \text{ops}^{128}_{\text{BlockId}} &\rightarrowtail 441348096 \\
+ \text{ops}^{128}_{\text{BlockId}} &\rightarrowtail 441348096 \\
+ \text{ops}^{128}_{\text{BlockId}} &\rightarrowtail 441348096 \\
\\
+ \text{ops}^{256,\text{stride=2}}_{\text{BlockPj}} &\rightarrowtail 955100160 \\
+ \text{ops}^{256}_{\text{BlockId}} &\rightarrowtail 439040000 \\
+ \text{ops}^{256}_{\text{BlockId}} &\rightarrowtail 439040000 \\
+ \text{ops}^{256}_{\text{BlockId}} &\rightarrowtail 439040000 \\
+ \text{ops}^{256}_{\text{BlockId}} &\rightarrowtail 439040000 \\
+ \text{ops}^{256}_{\text{BlockId}} &\rightarrowtail 439040000 \\
\\
+ \text{ops}^{512,\text{stride=2}}_{\text{BlockPj}} &\rightarrowtail 952817152 \\
+ \text{ops}^{512}_{\text{BlockId}} &\rightarrowtail 437885952 \\
+ \text{ops}^{512}_{\text{BlockId}} &\rightarrowtail 437885952 \\
\\
+ \text{ops}_{\text{GAvgPool}} &\rightarrowtail 102400 \\
+ \text{ops}_{\text{FullyC}} &\rightarrowtail 4096000 \\
\\
= 9299929600
\end{aligned}
$$

Last edited: January 2024

Declaration of Academic Integrity

1. I hereby confirm that this work — or in case of group work, the contribution for which I am responsible and which I have clearly identified as such — is my own work and that I have not used any sources or resources other than those referenced.

   I take responsibility for the quality of this text and its content and have ensured that all information and arguments provided are substantiated with or supported by appropriate academic sources. I have clearly identified and fully referenced any material such as text passages, thoughts, concepts or graphics that I have directly or indirectly copied from the work of others or my own previous work. Except where stated otherwise by reference or acknowledgement, the work presented is my own in terms of copyright.

2. I understand that this declaration also applies to generative AI tools which cannot be cited (hereinafter referred to as 'generative AI').

   I understand that the use of generative AI is not permitted unless the examiner has explicitly authorized its use (Declaration of Permitted Resources). Where the use of generative AI was permitted, I confirm that I have only used it as a resource and that this work is largely my own original work. I take full responsibility for any AI-generated content I included in my work.

   Where the use of generative AI was permitted to compose this work, I have acknowledged its use in a separate appendix. This appendix includes information about which AI tool was used or a detailed description of how it was used in accordance with the requirements specified in the examiner's Declaration of Permitted Resources.

   I have read and understood the requirements contained therein and any use of generative AI in this work has been acknowledged accordingly (e.g. type, purpose and scope as well as specific instructions on how to acknowledge its use).

3. I also confirm that this work has not been previously submitted in an identical or similar form to any other examination authority in Germany or abroad, and that it has not been previously published in German or any other language.

4. I am aware that any failure to observe the aforementioned points may lead to the imposition of penalties in accordance with the relevant examination regulations. In particular, this may include that my work will be classified as deception and marked as failed. Repeated or severe attempts to deceive may also lead to a temporary or permanent exclusion from further assessments in my degree programme.

Place and date

Signature