FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Code Generation for U-Nets Using the ExaStencils Code Generation Framework

Samuel Kemmler

Master Thesis

Code Generation for U-Nets Using the ExaStencils Code Generation Framework

Samuel Kemmler

Master Thesis

Aufgabensteller:	Prof. Dr. H. Köstler
Betreuer:	R. Angersbach, M. Sc.
Bearbeitungszeitraum:	1.9.2021 – 1.3.2022

Erklärung:

Erlangen, den 16. März 2022 . . . .

Contents

1 Introduction **7**

2 Related Work **9**

3 Theory **11**
3.1 U-Nets . . . 11
3.1.1 Architecture . . . 11
3.1.2 Training . . . 13
3.1.3 Convolutional Layer . . . 15
3.1.4 Rectified Linear Unit (ReLU) . . . 18
3.1.5 Pooling Layer . . . 18
3.1.6 Transposed Convolutional Layer . . . 19
3.2 Code Generation . . . 21
3.2.1 ExaStencils . . . 21
3.2.2 Abstract Syntax Trees (ASTs) . . . 24

4 Idea and Workflow **26**
4.1 Idea . . . 26
4.2 Workflow . . . 26

5 Implementation **28**
5.1 Variable Initialization . . . 28
5.2 Field Initialization . . . 31
5.3 Abstract Syntax Tree (AST) Building Process . . . 32
5.4 Layer 4 Code Printing . . . 37

6 Evaluation **39**
6.1 Correctness . . . 39
6.1.1 Application Scenario . . . 39
6.1.2 Correctness of Different Training Setups . . . 40
6.2 Usability and User Experience . . . 41
6.2.1 Training a Neural Network (NN) Using PyTorch . . . 41
6.2.2 Training a NN Using the GNNESS + ExaStencils Toolchain (GET) . . . 42
6.2.3 Comparison, Limitations, and Discussion . . . 43
6.3 Performance . . . 44
6.3.1 Technical Setup . . . 45
6.3.2 Code Generation . . . 47
6.3.3 Intra-Node OpenMP Parallelization (First Approach) . . . 48
6.3.4 Analysis of the Performance Issues of the First Approach . . . 52
6.3.5 Intra-Node OpenMP Parallelization (Second Approach) . . . 54
6.3.6 Double Precision . . . 59
6.3.7 Performance Analysis of the Second Approach . . . 60
6.3.8 Inter-Node Hybrid MPI-OpenMP Parallelization . . . 63
6.3.9 CUDA Parallelization for Graphics Processing Units (GPUs) . . . 64
6.3.10 Main Memory Usage . . . 66

7 Discussion and Conclusions **68**

8 Outlook **70**

Abstract

U-Nets have successfully been used for image segmentation tasks in recent years. One challenge in training Neural Networks (NNs) is long training times. However, the performance of deep learning frameworks like PyTorch is not always perfect since their implementations do not contain NN architecture or hardware-specific optimizations. Code generation technologies allow the automatic generation of codes optimized for a specific problem and hardware.
This thesis aims to implement a code generation toolchain that generates optimized and parallel C++ codes for given U-Nets. Ideally, the generated codes achieve a higher performance than the PyTorch implementation.
The user provides the U-Net architecture as a PyTorch model to the toolchain. The thesis implements a Python framework, which generates the training code in the Domain-Specific Language (DSL) ExaSlang for the given U-Net. Next, the ExaStencils code generation framework is used to generate parallel and optimized C++ code from the DSL code. This framework supports parallelization using OpenMP, hybrid MPI-OpenMP, and CUDA. The thesis provides performance measurements using a state-of-the-art CPU node (Intel Xeon Ice Lake) and GPU (NVIDIA A100). The performance for training U-Nets consisting of 5,879 to 368,955 trainable parameters using the toolchain is reported. For these sizes, the generated OpenMP parallel codes achieve a speedup between 1.8 to 25.4 compared with PyTorch on the entire node (72 cores). Hybrid MPI-OpenMP codes achieve a speedup of 10.4 to 14.9 on 16 nodes for 368,955 trainable parameters. The code generation times for the CPU codes are negligible. The generated CPU codes use less main memory than PyTorch. The performance of the generated CUDA parallel codes is worse than PyTorch, the code generation runtimes rise substantial for increasing U-Net sizes, and the generated codes use more GPU memory than PyTorch.
The thesis shows that code generation for U-Nets is possible using ExaStencils. For the examined U-Net sizes and Central Processing Units (CPUs), the thesis confirms the hypothesis that the generated codes perform better than PyTorch. This finding does not apply to the generated GPU codes.

Acknowledgements

First of all, I would like to thank my supervisors Prof. Dr. Harald Köstler and Richard Angersbach for their support throughout the thesis. In particular, the regular meetings answered a lot of my questions and have often given me new ideas. In this context, I would like to thank Sebastian Kuckuk for attending and enriching those meetings and supervising my thesis. When I had questions and posted them on Mattermost, they answered them immediately; thanks for that! In addition, I would like to specifically thank Richard Angersbach for his technical support regarding ExaStencils and the quick and straightforward problem solutions and implementations of new features in ExaStencils. Finally, I would like to thank the "Regionales Rechenzentrum Erlangen" RRZE for granting me access to their HPC clusters, especially having the Intel Xeon Platinum 8360Y CPU on the test cluster and the NVIDIA A100 on the TinyGPU cluster helped me a lot.

1 Introduction

In the last years, artificial NNs have become increasingly popular. Nowadays, they are used for a variety of problems. One application field are tasks where conventional algorithms fail or do not deliver satisfying results. Often, NNs even perform better than humans on specific tasks. A very successful and popular application field is image classification. Here, the task is to assign each image to one or more classes. For example, the "ImageNet Large Scale Visual Recognition Challenge" is a very well-known benchmark in the field of image classification. The challenge is to assign images containing one object to precisely one out of 1000 classes. The human top-5 accuracy is between 5.1%, and 12.0% [24]. The top-5 accuracy indicates how likely it is that one out of the first five classifications is correct. However, NNs achieve a top-5 accuracy of 99.02% [32] and thus outperform humans in this challenge. In addition, NNs are frequently used nowadays in speech recognition and translation. Another area in which NNs are used successfully is semantic image segmentation.
In contrast to classification, where the task is to assign the entire image to one or more classes, the task of semantic image segmentation is to assign each pixel to a class. This pixel-wise classification results in a division of the image into different segments. Fields of application are, for example, biomedical imaging or autonomous driving. A very successful publication (in terms of citations) in the field of deep learning is "Convolutional Networks for Biomedical Image Segmentation" [23]. This paper introduces a new NN architecture called U-Net, the de facto standard for semantic image segmentation. However, there are some challenges in training NNs. The first problem is that NNs often require large amounts of (often manually annotated) training data to achieve excellent results. Second, training NNs is computationally expensive and results in substantial training runtimes. This task can take up to several days or even weeks, depending on the NN size and the number of training data. If the training hyperparameters are optimized, one performs the training more than once, aggravating the problem and increasing the need for high-performance training codes. This thesis addresses the second problem. Within the scope of this thesis, the code generation framework ExaStencils generates highly optimized parallel codes for training U-Nets. The expectation is that the generated codes perform the training faster than the deep learning framework PyTorch. The reason for this expectation is that the code generation process incorporates knowledge of both the hardware and U-Net architecture, which is not possible to the same extent with deep learning frameworks like PyTorch.
The chair for computer science 10 at Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) develops and maintains a high-performance code generation framework called ExaStencils [13]. It comes with a DSL called ExaSlang [25] which was originally designed to efficiently write numerical PDE solvers like the multigrid method. However, as shown in the further course of this thesis, multigrid methods and training U-Nets are similar. Therefore, using ExaStencils in the context of training U-Nets is plausible. ExaStencils requires the algorithm described in the DSL ExaSlang to generate highly optimized and parallel C++ code. However, writing the training algorithm for U-Nets in ExaSlang is a time-consuming and repetitive task.
To not have to write the ExaSlang code manually for each U-Net, the thesis introduces a Python code generation framework named Generation of Neural Networks deployed as ExaSlang Specifications (GNNESS). Its task is to generate training codes in ExaSlang for given U-Nets. The user defines them in the form of PyTorch models. GNNESS and ExaStencils combined form a toolchain to generate C++ codes for training U-Nets given as PyTorch models. The thesis aims to provide a complete code generation toolchain for the generation of highly optimized and parallel C++ training codes for given U-Nets, which ideally reduces the training times compared to PyTorch. The toolchain should support OpenMP, hybrid MPI-OpenMP, and CUDA parallelization.
The thesis has the following structure. In section 2, relevant related work is discussed. In this context, the section summarizes publications regarding ExaStencils and different code generation approaches for NNs. Furthermore, it introduces a paper that inspired this thesis since it also uses a Python framework to generate ExaSlang code and then uses ExaStencils to generate C++ code. Hereafter, section 3 explains the most important theory necessary to understand the further course of the thesis. The theoretical foundations about the architecture, training (forward and backward pass), and layer types (convolution, ReLU, pooling, transposed convolution) used in U-Nets are provided. Furthermore, the section introduces the optimizers "Stochastic gradient descent" and "Adam" and presents the "Pixel-wise cross-entropy loss". In addition, it discusses the theory and functionality of the ExaStencils code generation framework and introduces ASTs. Section 4

describes this thesis' idea and how the workflow (i.e., code generation pipeline) looks like from the PyTorch model to the C++ training code. Section 5 explains implementation details of the GNNESS framework. It provides code snippets with detailed explanations regarding the following aspects of GNNESS: Initialization of ExaSlang fields and variables, building an abstract syntax tree representing the ExaSlang training code, and printing everything into a file. The subsequent section 6 evaluates the generated codes. It first verifies that the generated training codes are correct. Verification means that for the well-known Oxford-IIIT pet dataset, the section compares the loss values of the generated codes with the PyTorch implementation for different training setups and parameters. Furthermore, the usability and user experience of the GET are the content of this section. It addresses how training a NN using the GET differs from PyTorch from the user perspective. In addition, the section discusses the limitations of using the GET. Next, it presents the performance results. For different batch and NN sizes, it compares the performance of the generated codes to PyTorch. The section evaluates the performance for OpenMP parallel codes on a single Central Processing Unit (CPU) node, hybrid MPI-OpenMP parallel codes using multiple CPU nodes, and CUDA parallel codes using a GPU. Section 7 summarizes the results, discusses them, and provides conclusions. Finally, section 8 addresses future work.

2 Related Work

This section discusses related work that is either helpful in understanding the further course of this thesis or bringing its results into the proper perspective. First, this section discusses relevant literature regarding the code generation framework ExaStencils. Afterward, the publication that introduced U-Nets is summarized. Third, this section presents one publication with a similar code generation toolchain as in this thesis but for a different application. Finally, the section summarizes previous code generation approaches for training NNs. The subsections 3.1, 3.2.1, and 4 provide deeper insights into the topics presented below.

Code generation techniques allow application and hardware-specific code optimizations without manually implementing them. Especially when solving partial differential equations, efficient and scalable implementations are essential due to the high computational complexity. However, manually implementing and optimizing such solvers is very time-consuming. Furthermore, libraries have drawbacks since they often lack application and hardware-specific optimizations. The ExaStencils code generation framework [13] proposes a solution to this dilemma. It provides a domain-specific language called ExaSlang [25], which was designed for domain experts to specify numerical solvers (e.g., multigrid methods) conveniently and familiarly. ExaStencils then generates highly optimized parallel C++ code from the ExaSlang specification.

U-Nets are a particular Convolutional Neural Network (CNN) architecture first introduced by Ronneberger et al. [23] for the task of biomedical image segmentation. Since then, it has become one of the most successful (in terms of citations) publications in the field of deep learning. The authors are tackling the challenge of cell segmentation in light microscopy images for very few training samples. Their architecture consists of an encoder (i.e., contraction) and decoder (i.e., expansion) path. The encoder includes 3x3 convolutions, ReLUs, and 2x2 max poolings with stride 2. The decoder consists of 2x2 transposed convolutions with stride 2, 3x3 convolutions, and ReLUs. The characteristic element of U-Nets is "skip connections", which directly transfer high-resolution features from the encoding path to high-resolution levels of the decoding path. Therefore the data skips part of the NN. According to Ronneberger et al., successive convolution layers (after the skip connections) then learn to assemble more precise (less blurred) outputs using the information provided by the skip connections.

Generating complex codes using ExaStencils is the subject of the publication by Faghih-Naini et al. [9]. To solve the shallow water equations, they use a quadrature-free discontinuous Galerkin formulation combined with the analytical solution of element and edge integrals. They use the ExaStencils code generation framework to obtain highly optimized and parallel C++ code. For this purpose, they implemented a Python framework called Generation of Higher-Order Discretizations Deployed as ExaSlang Specifications (GHODDESS). Its purpose is to generate ExaSlang code for a given problem specification. This idea strongly inspires the approach taken in this thesis. The framework written for this thesis reused some of the code from GHODDESS.

Code generation for NNs has been investigated by several publications so far. One recent approach was presented by Venkat et al. [29]. They draw attention to the problem that NN training is time-consuming and requires efficient implementations. However, popular deep learning frameworks (e.g., TensorFlow or PyTorch) limit cross-layer optimizations. According to Venkat et al., these optimizations can improve performance by reducing data transfers. In their paper, they introduce a compiler called SWIRL. It generates high-performance CPU implementations for NNs. The NN architecture has to be specified using the domain-specific language LATTE. This way, they can generate fused, vectorized, and parallelized codes for CPUs only. The performance of their code is similar to Tensorflow using MKL-DNN.

Another NN code generation approach is introduced by Liu et al. [15]. They address the issue that the de facto standard hardware for training modern Convolutional Neural Networks (CNNs) are (server) CPUs or GPUs. According to Liu et al., Field-Programmable Gate Arrays (FPGAs) offer high performance, energy efficiency, and reconfigurability, making them suitable as CNN accelerators. Their paper introduces an automatic generator to generate Verilog HDL source code from a high-level hardware description language. In addition, they model the execution time a.o. for the code. They generate hardware designs with good performance for two famous CNN architectures (LeNet and AlexNet). Their approach leads to good performance while saving development time due to the code generation approach.

Furthermore, other kinds of NNs like rate-coded and spiking networks can be simulated using code

generation. Simulating them is the goal of Vitay et al. [30] in their paper "ANNarchy: a code generation approach to neural simulations on parallel hardware". They generate C++ code for CPUs (OpenMP) and GPUs (CUDA) with their neural simulator ANNarchy. They provide a high-level Python interface similar to PyNN. According to the authors, there is still room for performance improvements. Compared to other neural simulators, they achieve average performance.
The last publication described in this section is by Yavuz et al. [31]. They focus on simulating detailed brain circuit models in the context of computational neuroscience. These models are used to test hypotheses on brain functions. Yavuz et al. propose a code generation framework called GeNN (GPU-enhanced Neural Networks). Its goal is to use NVIDIA GPUs to simulate brain circuit models. For small networks, CPU simulations are comparable or even better than GPUs regarding performance. However, they achieve better performance on the Graphics Processing Unit (GPU) than on a single CPU core for most cases. The GPU was substantially slower than the CPU when switching to double-precision floating-point accuracy.

3 Theory

The subsequent section provides the theoretical foundations on which the thesis' further course is based. First, the subsection discusses the theory of U-Nets, mainly of which layers it consists. It also explains the training process of NNs in detail. The second part of this section deals with code generation. First, it introduces the ExaStencils code generation framework. Second, it presents ASTs, which are an essential concept in code generation since they provide abstract representations of code snippets.

3.1 U-Nets

The following subsection discusses the theory of NNs in general and U-Nets in particular. First, it describes and motivates the U-Net architecture. The subsection pays specific attention to the skip connections. Second, it introduces the four steps of one training epoch. Those are the forward pass, the loss computation, the backward pass, and finally, the optimization. The widely used "Pixel-wise cross-entropy loss" is explained. In addition, the subsection introduces two popular optimizers ("Stochastic gradient descent" and "Adam") in the context of optimization. Finally, it provides a detailed analysis of all computations used by the different U-Net layers in the forward and backward pass. Those layers are: convolution, ReLU, pooling, and transposed convolution.

3.1.1 Architecture

The following subsection about the U-Net architecture is based (unless otherwise stated) on the publication of Ronneberger et al. [23]. They introduced U-Nets in the context of biomedical image segmentation. A very well-known application area of NNs is image classification. Here, the task is training NNs to assign given images to one or many classes. For example, the ImageNet Challenge (ILSVRC) is a very famous NN benchmark that was introduced by Russakovsky et al. [24]. This contest aims to train NNs to assign images to one out of 1000 possible classes, given a set of training images. Unfortunately, image classification is not sufficient for many visual tasks. In the field of biomedical image processing, for example, the desired output should not be one class label per image (as in image classification). Instead, the task is to assign one class label to each pixel, i.e., performing a pixel-wise classification, which is also known as semantic segmentation [18]. Figure 1 illustrates the task that Ronneberger et al. are tackling.

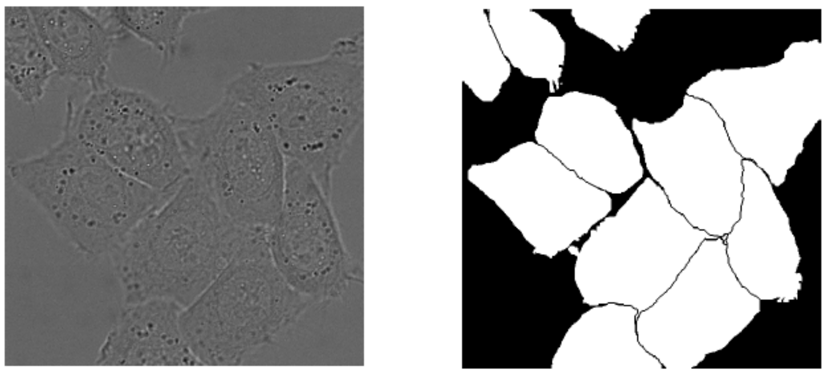


Figure 1: Image segmentation task. Raw image of cells on the left, segmentation mask on the right. Pixels that belong to the first segment (cells) are white, those who belong to the second segment (background) are black [23].

The task of the NN is to learn the mapping from a given raw image (left) to the segmentation mask (right), given a set of training images. So the NN should decide for each pixel whether it is part of a cell or the background. Another challenge that arises not only but also in biomedical image segmentation is that thousands of training images (as the ILSVRC provides) are usually not available. Ronneberger et al. introduced a new CNN called U-Net to address these two challenges. Figure 2 illustrates this architecture.

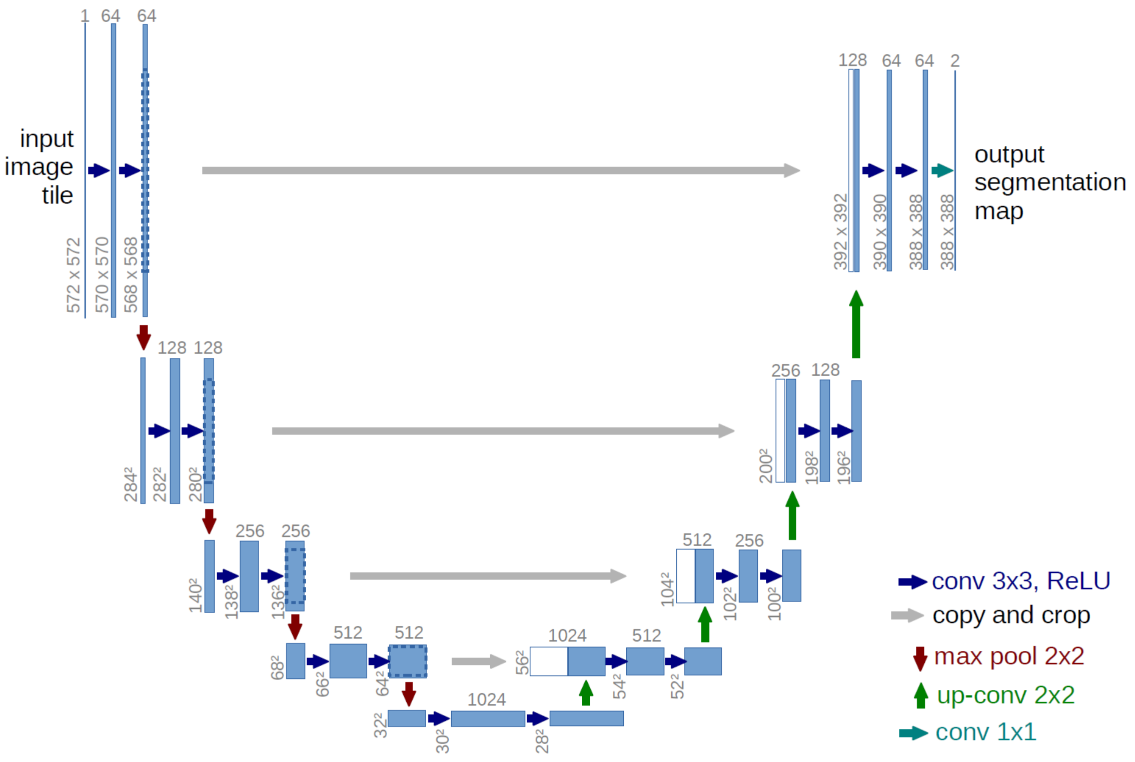


Figure 2: U-Net architecture introduced by Ronneberger et al. Blue boxes correspond to multi-channel feature maps. Grey numbers on top of the boxes represent the number of channels. The feature map sizes are denoted on the left of the boxes. Arrows indicate the data flow between different states, beginning at the input image tile and ending at the output segmentation map [23].

The U-Net architecture consists of a contracting path (left-hand side of Figure 2) and an expansive path (right-hand side of Figure 2). The contracting path consists of the typical operations of a CNN architecture used in image classification. Those are a concatenation of convolutions (kernel size: 3x3) followed by a ReLU and a pooling layer (size: 2x2, stride: 2x2). Since the convolutions do not use padding, the feature map size is reduced by 2 in each dimension per convolution. After each downsampling/pooling step, the number of feature channels doubles (starting at 64, ending at 1024). If this were an image classification task, typical architectures would append one or more fully connected layers to the contracting path, and the NN would be complete. Instead, the U-Net architecture adds an expansive path after the contraction path. It consists of transposed convolutions (kernel size: 2x2, stride: 2x2) for the upsampling, which halves the number of channels while doubling the feature map size. The transposed convolution is a form of reversed pooling. Two convolutions (kernel size 3x3) + ReLU follow on each transposed convolution. There is a feature-channel concatenation of the output of each transposed convolution and the feature map on the left-hand side of the gray arrows. This concatenation is indicated in Figure 2 using blue and white boxes. The name of these gray arrows is "skip connections" because the data skips some layers of the U-Net. The skip connections crop the data to the correct size since the feature map size on the left is bigger than on the right. Finally, there is a 1x1 convolution on the finest level. This convolution type is equivalent to applying a fully connected layer on each pixel in the channel dimension. This choice is reasonable since the desired output is a classification per pixel. The task of the contraction path is feature extraction. The transposed convolutions of the expansive path increase the feature map size, which is necessary for a segmentation task. According to Ronneberger et al., skip connections ensure that the successive convolutional layers can produce more precise (less blurred) outputs based on the high-resolution features.

3.1.2 Training

Training a NN in general and thus also a U-Net consists of four steps:

1. Forward pass
2. Loss computation
3. Backward pass
4. Optimization

This subsection discusses the four steps in detail below. Typically, these steps are not performed with all training data at once but only with a small subset, a so-called batch that usually contains a one or two-digit number of training samples. After completing the four training steps for the first batch, they are repeated for the remaining batches until all batches have gone through these four steps. When all batches have completed these steps, it is called an epoch. Typically, the training of a NN includes several epochs.
The so-called loss is a scalar quantity that expresses the difference between the actual and the expected output of the NN. The closer the output of the NN is to the desired output, the smaller the loss. Hence, minimizing the loss is equivalent to reducing the actual and expected output difference. Therefore, the only goal in training a NN is to minimize the loss for given training data by changing the trainable parameters (i.e., weights and biases). From a mathematical point of view, training a NN is nothing else than an optimization problem. Next, the first training step, the forward pass, is discussed.

Forward Pass

The training process begins with the forward pass. It is nothing else than handing over a batch to the NN and evaluating the output. The first layer gets the training batch as input. It calculates the output using its trainable parameters. The output of the first layer then serves as input for the second layer. This process repeats until the last layer has calculated its output. Afterward, the forward pass is complete. In Figure 2, the arrows represent the forward pass. A training batch starts on the left-hand side ("input image tile"). The input moves along the arrows during the forward pass, the respective layer carries out its computations, and finally, the last layer produces the output ("output segmentation map"). The individual operations performed at the arrows are discussed later in this subsection.

Cross Entropy Loss Function

The loss function computes a scalar value representing the accuracy of the NN prediction. The closer the actual output is to the expected output, the smaller the loss and vice versa. The loss calculation requires the output of the last layer of the NN and the corresponding expected output. In the case of image segmentation, the final layer of the NN must have as many channels as there are segments or classes for the pixels. For each pixel, the channel index with the highest value (after the last layer) indicates to which class the NN assigns this pixel. The loss calculation per pixel is performed as follows [6]:

$$loss(\mathbf{x}, class) = -log(\frac{exp(\mathbf{x}[class])}{\sum_j exp(\mathbf{x}[j])}) \tag{1}$$

where $\mathbf{x}$ is the vector containing all channel values of the pixel, *class* is the index of the correct class and j goes from 0 to $numClasses - 1$. The loss calculation repeats this computation for all pixels of a batch. The output of function 1 is "0" if the channel with the correct index has the value "1" and the other channel values are 0. Otherwise, it gets bigger the more the prediction deviates from the correct output.
However, since there are typically multiple pixels per image, numerous images per batch, and several batches, this value is typically averaged across all pixels of all batches to report a single value after each epoch. The backward pass (which the following subsection discusses) follows the loss calculation.

Backward Pass (Backpropagation)

The following subsection is based on [17]. The whole point of training a NN is minimizing the loss by changing the trainable parameters (i.e., weights and biases). The gradient of the loss with respect to (w.r.t) every trainable parameter is needed to optimize the loss. Getting those gradients is the only task of the backward pass. Its functionality is illustrated in Figure 3.

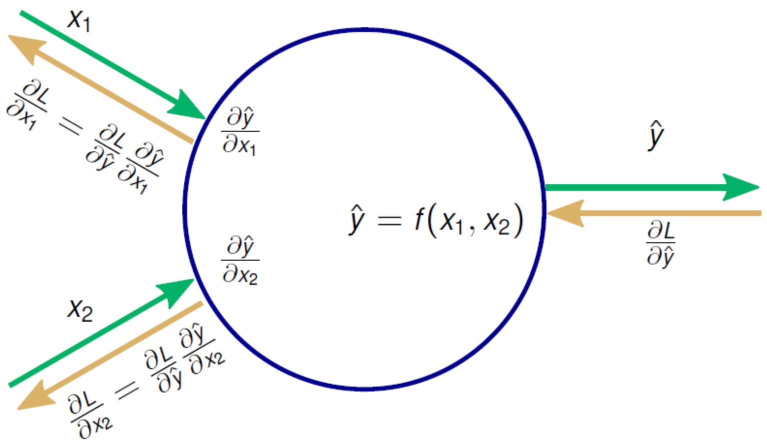

x_1
$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial x_1}$
$\frac{\partial \hat{y}}{\partial x_1}$
$\hat{y}$
$\hat{y} = f(x_1, x_2)$
$\frac{\partial L}{\partial \hat{y}}$
$\frac{\partial \hat{y}}{\partial x_2}$
x_2
$\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial x_2}$

Figure 3: Backpropagation algorithm for one NN layer. x_1 is the input of the layer, x_2 denotes its trainable parameters, $\hat{y}$ represents the output. The blue circle stands for one complete layer. L is the loss of the NN [16].

Figure 3 illustrates the data flow of the forward pass (green), as well as of the backward pass (ocher) for one layer of a NN. The output $\hat{y}$ is a function of the input x_1 and the weights x_2, which holds for any layer type (e.g., fully connected, convolutional). The forward pass receives x_1 from the previous layer, then $\hat{y}$ is computed using x_2 and passed on to the subsequent layer. In the backward pass, the layer receives the gradient of the loss w.r.t. the layer's output $\frac{\partial L}{\partial \hat{y}}$ (also known as error tensor) from the subsequent layer. Using this error tensor, the layer computes the gradient of the loss w.r.t. its own trainable parameters using the chain rule: $\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial x_2}$. Computing these gradients enables the upcoming optimization step to optimize the trainable parameters of this layer. Nevertheless, the backward pass is not yet complete. This layer's predecessor also requires the gradient of the loss w.r.t. the predecessor's output, which is nothing else than the gradient of the loss w.r.t. this layer's input. Using the chain rule this is $\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial x_1}$. In this way, every layer computes the gradient w.r.t. the own trainable parameters. The error tensor is propagated through the whole network, starting at the last and ending at the first layer. The loss computation layer computes the initial error tensor since it is at the end of the NN. The initial error tensor is the gradient of the loss function w.r.t its inputs (i.e., the output of the last layer). The idea of the backward pass is nothing else than recursively applying the chain rule.
After calculating the gradients of the loss w.r.t the trainable parameters in the backward pass, the final training step is to update the trainable parameters. An optimizer performs this update using the trainable parameters and the respective gradients. There exist various optimizers. For example, the very popular and simple Stochastic Gradient Descent (SGD) and more advanced Adaptive Moment Estimation (Adam) optimizers. GNNESS supports both.

SGD Optimizer

SGD is the simplest optimization algorithm. The update rule according to [7] is as follows:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha \cdot \nabla L(\boldsymbol{\theta}_{t-1}) \tag{2}$$

$\boldsymbol{\theta}_t$ is the updated vector of variables. In the context of NNs, $\boldsymbol{\theta}_t$ is the vector containing all trainable parameters (i.e., weights and biases). $\boldsymbol{\theta}_{t-1}$ is the old variable vector. α is the learning rate, which is the same for all variables and, therefore, a scalar value. L is the objective function (which is in the context of NNs the loss), and ∇L is the derivative of the objective function w.r.t. the variables. As previously mentioned, the backward pass yields $\nabla L(\boldsymbol{\theta}_{t-1})$ for trainable parameters (= variables to be optimized).

Adam Optimizer

The Adam optimizer is significantly more complex than SGD. The following section is based on the article by Kingma et al. [12] which first introduced Adam. Unlike SGD, for each variable of the optimization problem, an individual adaptive learning rate is computed from estimates of the first and second gradient moments.
In the first step of Adam's update, the biased first and second moment estimates $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$ are updated:

$$\boldsymbol{m}_t = \beta_1 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_1) \cdot \nabla L(\boldsymbol{\theta}_{t-1}) \tag{3}$$

$$\boldsymbol{v}_t = \beta_2 \cdot \boldsymbol{v}_{t-1} + (1 - \beta_2) \cdot [\nabla L(\boldsymbol{\theta}_{t-1})]^2 \tag{4}$$

Initially, $\boldsymbol{m}_0 = \boldsymbol{0}$ and $\boldsymbol{v}_0 = \boldsymbol{0}$. Again, ∇L is the derivative of the objective function w.r.t. the variables and $\boldsymbol{\theta}_{t-1}$ is the old variable vector. $\beta_1, \beta_2 \in [0, 1)$ denote the exponential decay rates for the moment estimates.
Second, the bias-corrected first and second moment estimates $\hat{\boldsymbol{m}}_t$ and $\hat{\boldsymbol{v}}_t$ are computed:

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{(1 - \beta_1^t)} \tag{5}$$

$$\hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{(1 - \beta_2^t)} \tag{6}$$

Finally, the vector of variables $\boldsymbol{\theta}_t$ is updated using the previously calculated moment estimates:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \frac{\alpha \cdot \hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon} \tag{7}$$

$\boldsymbol{\theta}_{t-1}$ is the old vector of variables, and α is the scalar step size. Kingma et al. propose the following default settings: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^-8$. They empirically demonstrate that Adam compares favorably to other optimization methods. Since then, Adam has become one of the most common optimizers in deep learning.

3.1.3 Convolutional Layer

The following subsection about convolutional layers is based (unless otherwise stated) on [16]. For the sake of simplicity, the subsection assumes a convolutional layer without bias. The bias only plays a minor role in the calculation complexity. Figure 4 illustrates the convolutional layer forward pass.

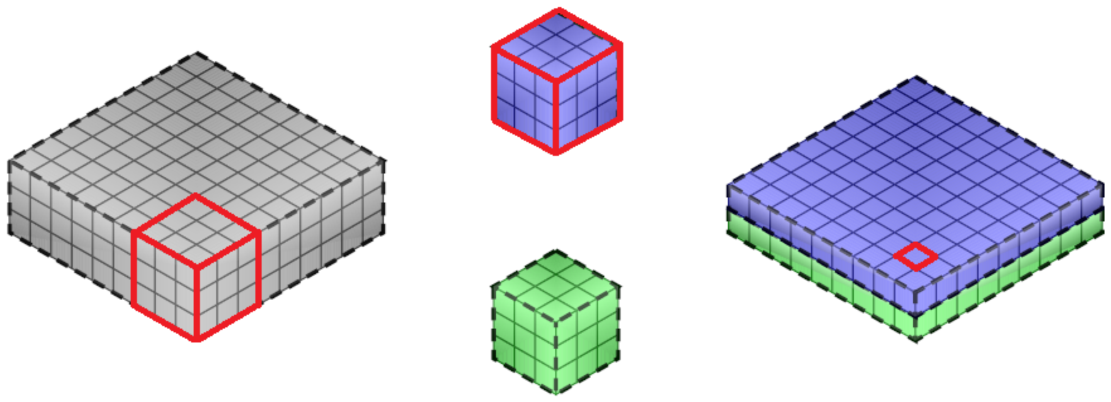


Figure 4: Forward pass of a convolutional layer. Input on the left, kernels in the center, output on the right. The meaning of the red lines is explained below (modification of [16]).

Figure 4 illustrates a convolutional layer that receives an input image with three feature channels and a resolution of 9x9 and generates an output image with two feature channels and a resolution of 9x9. The two cubes in the center each represent one kernel. Each of them is responsible for generating one feature channel of the output image, hence the corresponding coloring. The following example describes the computation of one output pixel using the input image and the associated kernel. The red pixel of the blue output channel in Figure 4 is obtained by multiplying the blue kernel element-wise with the red cube of the input image and adding up the results. In total, this requires 27 multiplications and 26 additions per output pixel. For the calculations of the remaining output pixels of the blue channel, the process is the same for every pixel, but with an accordingly shifted input tensor cube. This process is repeated for the green output channel but with the green kernel. The input must be padded with one pixel row on each side to calculate the boundary output pixels.
Let the input size be (X, Y, S), where (X, Y) is the image resolution, and S is the number of input channels. Then the kernels have to be of size (M, N, S) where the kernel size (M, N) has to be specified by the user. Typical kernel sizes are in the range between 2 and 5. If there are H kernels, the output is of size (X, Y, H) (assuming "same" padding, which means padding in x and y-direction such that the convolution preserves the image resolution). Note that the number of trainable convolution parameters does not depend on the image size. In this way, the training uses much more training data per weight than using a fully connected layer. The following code snippet presents one out of many possible pseudocodes for the convolutional layer forward pass.

```
1 loop over output image {
2   for number of images per batch {
3     for number of channels per image {
4       Compute one output pixel by multiplying the kernel $channel by the
            corresponding cube of the input and adding up the results
5     }
6   }
7 }
```

Listing 1: Pseudocode for the convolution forward pass.

The thesis indicates line numbers using brackets in the further course. The outer loop (1) iterates over all pixels of the output image, the middle loop (2) iterates over all images of the batch (see subsection 3.1.2), and the innermost loop (3) iterates over all channels of the output image.
Figure 5 illustrates the first part of the backward pass, the gradient w.r.t. the previous layer. Subsection 3.1.2 provides details on the different parts of the backward pass.

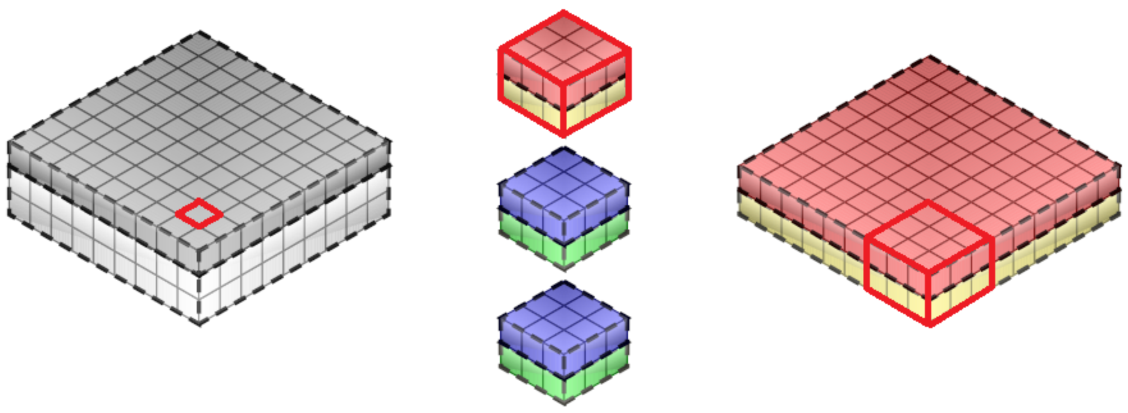


Figure 5: First part of the backward pass of a convolutional layer. Gradient w.r.t. the output of the previous layer. New error tensor on the left, kernels in the center, old error tensor on the right. The meaning of the red lines is explained below (modification of [2]).

The first part of the backward pass of a convolutional layer calculates the gradient w.r.t. its input, also called new error tensor, which the layer afterward passes to the previous layer. The gradient w.r.t. the output of this layer, also known as the old error tensor, is provided by the

subsequent layer. The most important message is that computing the gradient w.r.t. the previous layer's output is similar to the forward pass. The computation of the gradient w.r.t. the previous layer applies the kernels to the old error tensor to compute the new error tensor. Figure 5 represents the backward pass for the same layer whose forward pass Figure 4 presents. This layer has three input and two output channels, so the new error tensor has three, and the old error tensor has two channels. Therefore, three kernels are required, each with two feature channels. The kernels of the backward pass are obtained by reassembling the kernels of the forward pass. For example, the upper backward kernel in Figure 5 is assembled by stacking up the upper channels of both forward kernels from Figure 4. The second backward kernel is generated by stacking up the middle channels of both forward kernels. The red lines in Figure 5 indicate which kernel is multiplied by which cuboid of the old error tensor to calculate one pixel of the new error tensor. For more details, see the description of the forward pass. Since the first part of the backward pass is very similar to the forward pass, no pseudocode is presented.
Figure 6 illustrated the second part of the backward pass, the gradient of the loss w.r.t. the trainable parameters.

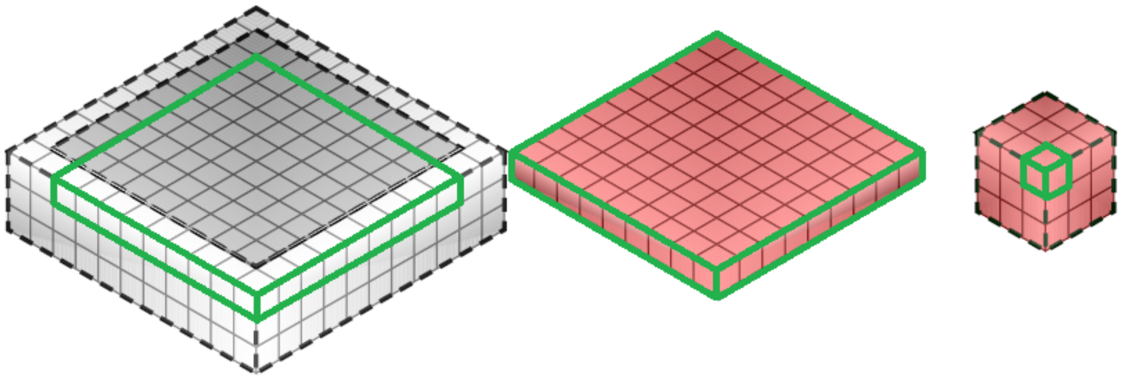


Figure 6: Second part of the backward pass of a convolutional layer. Gradient w.r.t. the trainable parameters. Input image on the left (with one padding row on each side in x and y direction), error tensor in the center, gradient w.r.t. the trainable parameters on the right. The meaning of the green lines is explained below (modification of [2]).

The second part of the backward pass of a convolutional layer computes the gradient w.r.t. the trainable parameters of the layer. Those are only the kernel weights in this example since there are no biases. The result of this procedure must have the same shape as the kernel such that there is one gradient per trainable parameter in the end. The computation of the gradient w.r.t. the trainable parameters uses the old error tensor (center) and the input tensor of the forward pass (left-hand side). On the right-hand side of Figure 6, the gradient w.r.t. one of the two kernels is illustrated. Its calculation is again a convolution, in this case between the input tensor and the old error tensor. The green value of the gradient cube is computed by multiplying the old error tensor with the green area of the padded input tensor and summing up the results, just as in the forward pass. Using the same padding as in the forward pass (in this case, one row in x and y direction), the resulting gradient w.r.t the kernel has the correct size in x and y direction (in this case 3x3). To get the gradient w.r.t the second kernel, this is repeated using the second channel of the error tensor (not shown in Figure 6). The pseudocode below illustrates one out of many possible pseudocodes for the convolution forward pass.

```
loop over error tensor {
  for number of images per batch {
    for number of kernels {
      for number of channels per image {
        for number of elements per kernel in x and y direction {
          Multiply one element of the error tensor by one element of the input
              tensor and add the result to the gradient w.r.t. the corresponding
              weight.
        }
      }
```

```
        }
    }
}
```
Listing 2: Pseudocode for the convolution backward pass.

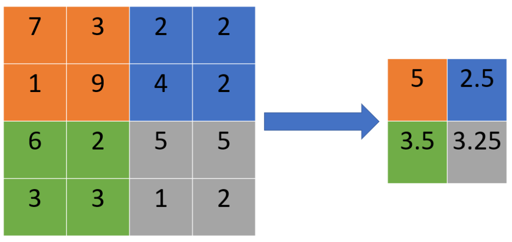
The outer loop (1) iterates over all pixels of the error tensor, the second loop (2) iterates over all images of the batch, the third loop (3) iterates over all kernels, the fourth loop (4) loops over all channels of the input image and the innermost loop (5) iterates over all kernel elements in x and y direction (9 in case of Figure 6). The innermost loop executes one multiplication and "+=" operation.
The most important message is that the forward pass and both parts of the backward pass are convolutions, respectively stencil operations.

3.1.4 ReLU

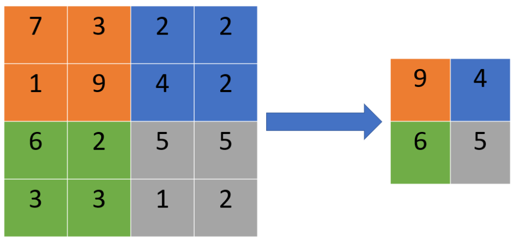
The ReLU layer is relatively simple. In the forward pass, each pixel of a batch with a positive value remains unchanged, and it sets all negative values to 0. In the backward pass, the ReLU layer sets those pixels of the error tensor that were negative in the forward pass to 0. The other pixels remain unchanged.

3.1.5 Pooling Layer

The following subsection about the pooling layer is based (unless otherwise stated) on [16]. The purpose of pooling layers is to compress and aggregate information across spatial locations (i.e., creating a "summary" of a region), decrease the image resolution and reduce overfitting. However, this assumes that the features are hierarchically structured. In addition, the exact feature location must not be significant since pooling provides translational invariance. The most popular forms are average and max pooling. This subsection describes both versions. Figure 7 illustrates the computations happening in the forward pass of an average and max-pooling layer.



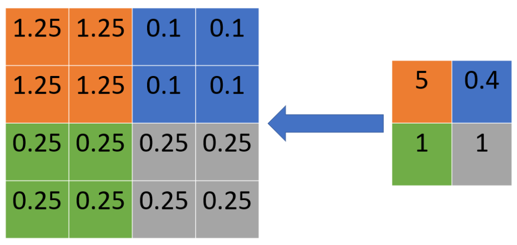
(a) Average pooling (kernel size: 2x2, stride: 2x2) forward pass.



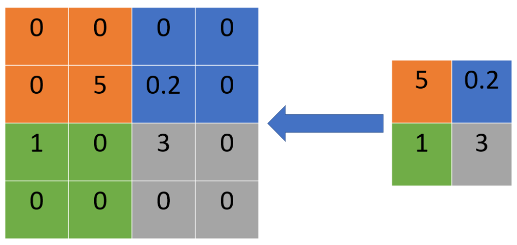
(b) Max pooling (kernel size: 2x2, stride: 2x2) forward pass.

Figure 7: Forward pass of an average and max pooling layer.

The average pooling forward pass averages over the filter window elements. In the previous example, the kernel shape and, therefore, the filter windows size is 2x2. Figure 7 marks the pixels that belong to the same filter window with the same color. Afterward, the pooling layer writes the average into the corresponding output pixel (indicated with the same color as the filter window), and then the filter window is shifted by "stride" elements (in this case 2).
The max-pooling forward pass works on the same principle as the average pooling forward pass except that instead of averaging over the filter window elements, their maximum is computed.
Figure 8 illustrates the backward pass of an average and max-pooling layer.



(a) Average pooling (kernel size: 2x2, stride: 2x2) backward pass.



(b) Max pooling (kernel size: 2x2, stride: 2x2) backward pass.

Figure 8: Backward pass of an average and max pooling layer.

The average pooling backward pass divides each incoming old error tensor element by the number of filter window elements (in the previous example 4) and writes the result into all elements of the corresponding filter window. As in the forward pass, Figure 8 marks the element of the old error tensor and the associated filter window elements in the same color.
The max-pooling backward pass differs substantially. It copies the element of the old error tensor into a single pixel of the filter window. It uses the filter window pixel with the maximum value in the forward.

3.1.6 Transposed Convolutional Layer

The purpose of the transposed convolutional layer is to upsample the image resolution. It is, therefore, the pooling layer counterpart. In the following, for the sake of simplicity, the transposed convolutional layer is described for an input and output image with only one channel. The following subsection assumes a transposed convolutional layer without bias. As with "normal" convolutions, transposed convolutions can also be used in U-Nets if the number of input and output channels differ and is higher than one. Figure 9 illustrates the forward pass of a transposed convolutional layer.

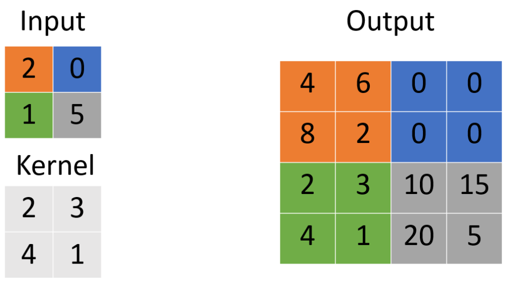


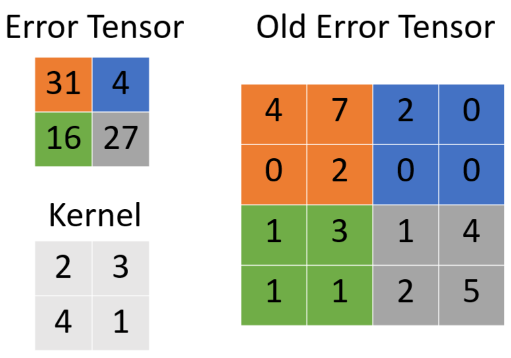
Figure 9: Forward pass of a transposed convolutional layer (kernel size: 2x2, stride: 2x2).

The transposed convolution forward pass multiplies each input image pixel by the whole kernel. In Figure 9, the input size is 2x2. Since the kernel size is 2x2 and the stride is 2x2, the output is 4x4. Figure 9 marks output and input pixels with the same color if the input pixel is used to compute the output pixels. For example, the transposed convolution obtains the four orange output pixels by multiplying each kernel element by the orange input pixel (= 2). Since the kernel size and striding are both 2x2, there is no overlapping of the colored regions of the output image. The following pseudocode shows one out of many possible implementations of the transposed convolution forward pass with a kernel size of 2x2 and a stride of 2x2.

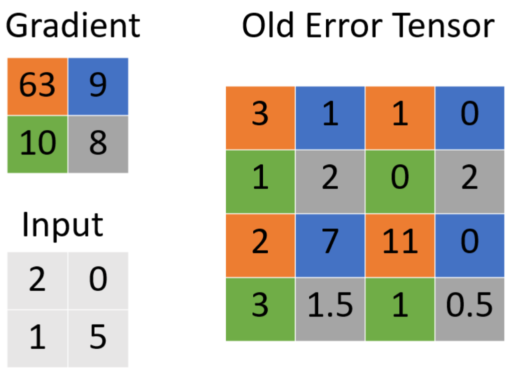
```
loop over output image {
  for number of images per batch {
    if ( Both coordinates of the current pixel are even ) {
      Compute one output pixel by multiplying the top left kernel value by the
          corresponding input value
    }
    if ( The first coordinate is even, the second is odd ) {
      Compute one output pixel by multiplying the top right kernel value by the
          corresponding input value
    }
    if ( The first coordinate is odd, the second is even ) {
      Compute one output pixel by multiplying the bottom left kernel value by the
          corresponding input value
    }
    if ( Both coordinates of the pixel are odd ) {
      Compute one output pixel by multiplying the bottom right kernel value by
          the corresponding input value
    }
  }
}
```

Listing 3: Pseudocode for the transposed convolution forward pass.

The outer loop (1) iterates over all pixels of the output image, the inner loop (2) iterates over all batch images. Depending on its coordinates, the transposed convolution multiplies the corresponding input element by one kernel value for each output pixel.
In Figure 10, the transposed convolution backward pass is shown. As with the "normal" convolutions, it consists of two parts. First, calculate the gradient w.r.t. the previous layer. Second, compute the gradient w.r.t. the weights.



(a) Gradient w.r.t. the previous layer.



(b) Gradient w.r.t. the weights.

Figure 10: Backward pass of a transposed convolutional layer (kernel size: 2x2, stride: 2x2). Error tensor denotes the new error tensor.

The first part of the backward pass computes the gradient w.r.t. the previous layer's output, which means computing the new error tensor (passed to the previous layer) using the old error tensor (provided by the subsequent layer) and the kernel. For each element of the new error tensor, the transposed convolution multiplies the kernel with four elements of the old error tensor. Those are the same ones that resulted from this input pixel in the forward pass (compare Figure 9 and 10a). Figure 10 marks new error tensor pixels and old error tensor elements with the same color if the old error tensor elements is used to compute the new error tensor pixels. The following pseudocode shows one out of many possible implementations for the transposed convolution backward pass (gradient w.r.t. the lower layer).

```
loop over new error tensor {
  for number of images per batch {
    Compute one output pixel by multiplying the kernel element−wise with the
        corresponding area of the old error tensor and adding up the results
  }
}
```

Listing 4: Pseudocode for the transposed convolution backward pass (gradient w.r.t. the lower layer).

The outer loop (1) iterates over all pixels of the new error tensor, the inner loop (2) iterates over all batch images. Depending on the pixel coordinates, the corresponding elements (see color

coding in Figure 10) of the old error tensor are element-wise multiplied with the kernel and added up. The transposed convolution carries out these computations for each pixel of the batch.
In the second part of the backward pass, the gradient w.r.t. the weights is computed using the old error tensor (provided by the subsequent layer) and the input image (provided by the previous layer in the forward pass). Each gradient element computation is done by multiplying the input by those elements of the old error tensor that were computed using the corresponding weight in the forward pass. Again, the colors indicate which elements of the old error tensor the transposed convolution multiplies with the input image to calculate which gradient elements. One out of many possible pseudocodes for the transposed convolution backward pass (gradient w.r.t. the weights) is presented below.

```
1 loop over error tensor {
2   for number of images per batch {
3     for number of gradient elements {
4       Compute one gradient element by multiplying the input image element-wise by
            the corresponding pixels of the old error tensor and summing up the
            results
5     }
6   }
7 }
```

Listing 5: Pseudocode for the transposed convolution backward pass (gradient w.r.t. the weights).

The outer loop (1) iterates over all pixels of the error tensor, the inner loop (2) iterates over all batch images. Depending on the coordinates of the old error tensor element, it is multiplied by the corresponding input element (see color coding in Figure 10b), and the result is added to the corresponding gradient element. After explaining the necessary theory of U-Nets, the following subsection about code generation discusses the ExaStencils framework and introduces ASTs.

3.2 Code Generation

This subsection covers the topic of code generation. First, it presents the ExaStencils code generation framework used in this thesis. Second, a brief introduction into ASTs is given. They are an essential part of code generation since they allow abstract representations of code snippets.

3.2.1 ExaStencils

The following subsection about ExaStencils [26] is based (unless otherwise stated) on the publication of Kuckuk [13]. This section is a high-level description of ExaStencils. For more details, see [13]. The chair for computer science 10 at FAU has its research focus on system simulations. Simulating physical systems usually involves the solution of partial differential equations. Solving them is a computationally intensive task for large problem sizes. Hence, this requires highly optimized and parallel codes that fully exploit the power of a high-performance compute node/cluster. There are two ways to deal with this problem: writing code and optimizing it manually for the specific problem and hardware or using libraries like Deal.II [1], etc. However, the first approach has the disadvantage that these codes are hardly reusable, and this approach is therefore very time-consuming. The second option does not have this problem, but the implementations of the libraries the are often not optimal since these libraries were designed neither for specific problems nor hardware. To counter this dilemma, the chair for computer science 10 at FAU developed and maintains the ExaStencils code generation framework [13]. It uses the approach of providing a domain-specific language (DSL) combined with a code generator to automatically generate parallel codes optimized for the specific problem and hardware specification. To be precise, ExaStencils can generate code for shared-memory parallelism using OpenMP, distributed memory parallelism using MPI, and GPUs using CUDA. The further course of this subsection describes the workflow of ExaStencils first. Second, it describes all files the user has to provide to generate code with ExaStencils.

Workflow

The user has to provide the problem specification/algorithm in a DSL called ExaSlang [25] to generate code using ExaStencils. ExaSlang provides a convenient syntax for describing partial differential equation problems and corresponding numerical solvers. ExaSlang is a hierarchical multi-level language, which means that it offers four abstraction levels in which the user can specify the problem/solver. This multi-level approach is valuable because different users have various expectations and needs. Some users want to specify the problem/solver abstractly without details regarding the solution implementation or domain decomposition. On the other hand, some users have specific ideas regarding solver implementations and domain decompositions. Figure 11 illustrates the layered structure of ExaSlang.

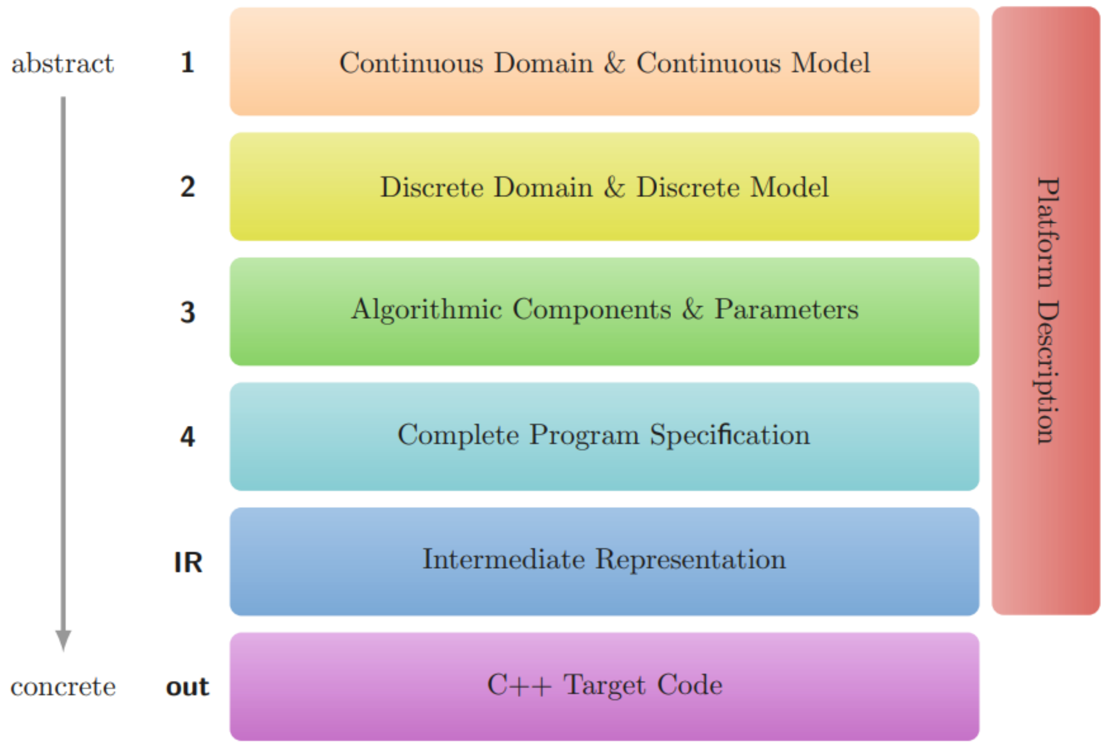

abstract
concrete
1
Continuous Domain & Continuous Model
2
Discrete Domain & Discrete Model
3
Algorithmic Components & Parameters
4
Complete Program Specification
IR
Intermediate Representation
out
C++ Target Code
Platform Description

Figure 11: ExaSlang layers of abstraction [14].

Figure 11 illustrates the different layers of abstraction that appear in the context of ExaStencils. Levels 1 - 4 are the previously described layers of ExaSlang. ExaStencils internally transforms the ExaSlang specification into an intermediate representation. Using the platform description (i.e., hardware details), ExaStencils finally generates C++ target code.
Additionally, the user must provide a knowledge file containing all code generation parameters (e.g., parallelization and domain composition). ExaSlang layer 1 code is most appropriate if the user is interested in specifying a continuous problem (without caring about the discretization, solver, and parallelization) in a latex-like syntax. If the discretization is of interest, but the solver implementation details and the parallelization are not, then layer 2 should be chosen. If the solver and its parameters are additionally of interest but not all details of the discretization, layer 3 is the right choice. If the user wants maximum flexibility to implement the full problem/solver, the user can provide a complete program specification as layer 4 code. Since this thesis uses the layer 4 code, the following subsection explains it in more detail. Additionally, the upcoming paragraphs describe the previously mentioned platform and knowledge file.

Layer 4 Code

The layer 4 code is the lowest possible abstraction level on which the user can specify the problem/-solver. Many concepts from high-level programming languages are part of the layer 4 code. This thesis uses ExaStencils for training NNs although it was not designed for this purpose. Therefore, using layer 4 code for specifying the training algorithm is most appropriate since layer 4 provides the highest flexibility required to describe algorithms other than PDE solvers. The following code snippet shows a minimal layer 4 code example with its main components. Underneath is a description of the different parts of the code.

```
 1 Domain global < [0, 0] to [1, 1] >
 2
 3 Layout Image1Layout < Matrix<Real, 8, 1> , Cell > @9 {
 4   innerPoints     = [ 512, 512 ]
 5 }
 6
 7 Layout Image1Layout < Matrix<Real, 8, 1> , Cell > @8 {
 8   innerPoints     = [ 256, 256 ]
 9 }
10
11 Field Image1_matrix < global, Image1Layout, None > @(9, 8)
12
13 Function Test@9 (  ) {
14   loop over Image1_matrix {
15     Image1_matrix[0][0] = 0.0
16   }
17   print("Test")
18 }
19
20 Function Application (  ) {
21   initGlobals (  )
22   initDomain (  )
23   initGeometry (  )
24   startTimer ( 'test timer' )
25   repeat 5 times  {
26     Test@9 (  )
27   }
28   stopTimer ( 'test timer' )
29   printAllTimers (  )
30   destroyGlobals (  )
31 }
32
33 Globals {
34   Var loss  :  Real  =  0.0
35   Var num_pixels : Int = 262144
36   Var m : Matrix<Real, 2, 2> = {{0, 1}, {2, 3}}
37 }
```

Listing 6: Minimal layer 4 code example.

The first part of the layer 4 code above defines the domain size (1), relevant for PDE solvers but not for this thesis. Afterward, layouts are defined (3 - 9). They describe the size and properties of the field data structures defined below. The first layout (3 - 5) describes a 2D array-like data structure of size 512x512. Every cell of this field contains a matrix with 8x1 floating-point values. Layouts and fields are defined for a specific level (indicated by the @ symbol). The idea of defining data structures for certain levels stems from multigrid solvers. However, this concept also appears in the context of U-Nets due to the resolution changes in the encoder/decoder path. The second layout with the same name (7 - 9) describes the same fields but on a coarser level with 256x256 elements. In the following code section (11), a field data structure is defined on two levels using the previously mentioned layouts. Next, a Test function is defined that iterates over the previously defined field. For every field cell, the statement inside the loop sets the matrix element [0][0] of this cell to 0. Again, the @ symbol indicates that this function operates on data structures of the given level (in this case, 9). Afterward, the Application function is defined. This function is the main routine. It contains some necessary function calls (initGlobals, initDomain, initGeometry, printAllTimers, destroyGlobals), which this subsection does not describe in detail. In addition, the code starts (24) and stops (28) a timer and calls the Test function inside a repeat statement (= for loop). Finally, the Globals section is defined. The user can use it to define global variables. The example above defines one floating-point (= Real), one integer, and one floating-point matrix variable. This thesis has to implement the whole training process using this syntax.

Knowledge File

The knowledge file specifies all parameters describing the implementation of the generated code. The following code snippet shows such a file.

```
dimensionality                  = 2

minLevel                        = 8
maxLevel                        = 9

domain_onlyRectangular          = true
domain_rect_generate            = true
domain_rect_numBlocks_x         = 1
domain_rect_numBlocks_y         = 1
domain_rect_numBlocks_z         = 1
domain_rect_numFragsPerBlock_x  = 2
domain_rect_numFragsPerBlock_y  = 1
domain_rect_numFragsPerBlock_z  = 1

mpi_enabled                     = false

omp_enabled                     = true
omp_numThreads                  = 4
omp_parallelizeLoopOverFragments = false
omp_parallelizeLoopOverDimensions = true
omp_minWorkItemsPerThread = 1
useDblPrecision                 = false
```

Listing 7: Knowledge file example.

The previous example states the data structures' dimensionality first. In addition, it defines the minimum and maximum levels of the fields. Furthermore, it describes the domain decomposition into blocks and fragments. The knowledge file also contains information on which kind of parallelization should be used (OpenMP, MPI, Cuda). Additionally, it provides information on whether to use 32-bit or 64-bit floating-point precision data structures. In addition, ExaStencils supports plenty of code optimizations, which the user can specify in the knowledge file. Those are, for example, vectorization, address precalculation, and common subexpression elimination. In total, the user can define more than 200 parameters in the knowledge file.

Platform File

The last file ExaStencils needs for code generation is the platform file. It contains information about the hardware and software of the target platform. This information enables ExaStencils to perform platform-specific code optimizations like blocking for specific memory sizes and vectorization using the supported SIMD instructions. The following code snippet shows a brief platform file example.

```
targetOS                      = "Linux"
targetCompiler                = "GCC"
targetCompilerVersion         = 8
targetCompilerVersionMinor    = 1
simd_instructionSet           = "AVX512"
```

Listing 8: Platform file example.

Given the platform file above, ExaStencils generates SIMD vectorized code using AVX512 intrinsics if the optimization flag "opt_vectorize = true" is used in the knowledge file. In addition, the generated Makefile uses the GCC compiler, and it is assumed that at least GCC 8.1. will be used for the compilation.
The following subsection describes ASTs, which play an essential role in code generation.

3.2.2 ASTs

The following subsection about ASTs is based (unless otherwise stated) on [33]. An AST is a tree data structure used to represent a code's syntax structure. Since they do not include all source code details, ASTs are called abstract. They contain nodes that correspond to constructs or symbols of the corresponding code snippet. ASTs neglect syntactic elements like punctuations. Instead, they represent the code's lexical information and syntactic structure. ASTs typically have one root which represents the whole code snippet. Every node except the root node has one parent node and one or more children, which describe the syntactic structure of the parent node. Nodes without children are called leafs and contain indivisible operations like variable accesses. The following code snippet is taken from [27] and represents a simple method called m.

```
void m(){
   if(c>2)
      x=c;
   while(c<10){
      x+=p();
      c++;
   }
}
```

Listing 9: Definition of the function m.

The code snippet consists of the method "void m()" encompassing two statements: if and while. Both again consist of several statements. Figure 12 illustrates the corresponding AST.

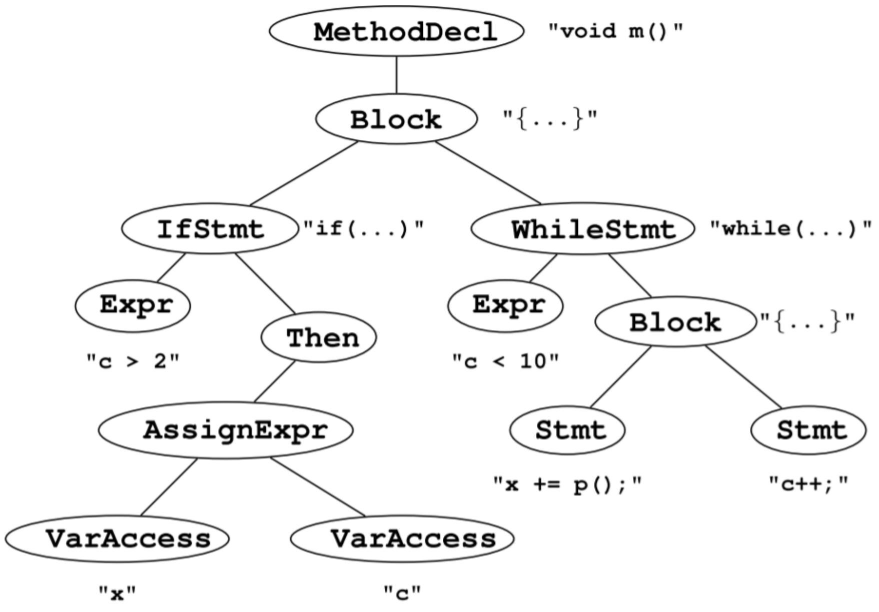


Figure 12: AST of the previously defined method m() [27].

The node MethodDecl is located at the top of Figure 12 and represents the method declaration. Since the code snippet only consists of this method, the corresponding node is the root. Its body consists of the if statement and the while loop. Therefore, the root node has two children, the IfStmt and the WhileStmt. They both consist of an expression/condition and a body. This logic is continued recursively until the indivisible operations, and thus the leaf nodes are reached. Code generators use ASTs to represent, modify and finally print code.

4 Idea and Workflow

The following subsection first introduces the idea of this thesis and then describes the workflow, i.e., the GET.

4.1 Idea

Typically, training NNs is done using libraries such as TensorFlow or PyTorch. Using them, however, leads to the same problem that Kuckuk [13] describes concerning PDE solver libraries: they omit NN architecture-specific optimizations, and they do not gear the implementation for specific hardware. ExaStencils provides a solution to this problem by providing a DSL combined with a code generator as described in subsection 3.2.1. Using code generation technology for generating U-Net training codes is the idea of this thesis. In this way, the code generator can optimize the training code for a specific NN architecture and hardware. Additionally, this enables cross-layer optimizations as described by [29] which reduces data transfers (see section 2). The question arises in this thesis, which code generation framework to use for generating NNs training codes. ExaStencils is a framework to generate highly optimized and parallel codes for PDE solvers like the multigrid method. Nevertheless, it is suitable for generating U-Net training codes. Table 1 shows the reasons for this.

Multigrid	U-Net
Smoothing (= apply stencil)	Convolution
Restriction	Pooling
Prolongation	Transposed convolution

Table 1: Comparison between multigrid algorithms and U-Nets. Each line contains a similar operation in the terminology of multigrid (left) and U-Nets (right).

The left column shows the essential operations of multigrid algorithms. They consist of smoothing, restriction, and prolongation. The most relevant operations of U-Nets are convolution, pooling, and transposed convolution. In Figure 1, the operations in the same row are very similar, so smoothing is almost the same as convolution, the restriction is similar to pooling, and the prolongation is similar to transposed convolution. This similarity holds for both the forward and backward pass, i.e., the full training. In summary, multigrid algorithms are very similar to the training of U-Nets. Therefore, it makes sense to use ExaStencils for generating U-Net training codes.

4.2 Workflow

As mentioned in subsection 3.2.1, ExaStencils receives ExaSlang code from the user and generates C++ code from it. Since ExaStencils originally is designed to generate PDE solver codes but not U-Net training codes, using layer 4 of ExaSlang is reasonable since layer 4 offers maximum flexibility. This flexibility is essential when using ExaStencils for such a different task. However, writing a U-Net training algorithm manually in layer 4 code is time-consuming, error-prone, and repetitive, especially for large U-Nets. Therefore, this thesis uses code generation technology to generate layer 4 code. For this purpose, this thesis introduces an ExaSlang layer 4 code generation framework written in Python. The framework receives a U-Net as a PyTorch model file and generates the corresponding training algorithm in layer 4 code. In addition, the code generation framework receives a "config" file that contains all training hyperparameters like the loss function, optimizer, and learning rate. Figure 13 illustrates the complete workflow.

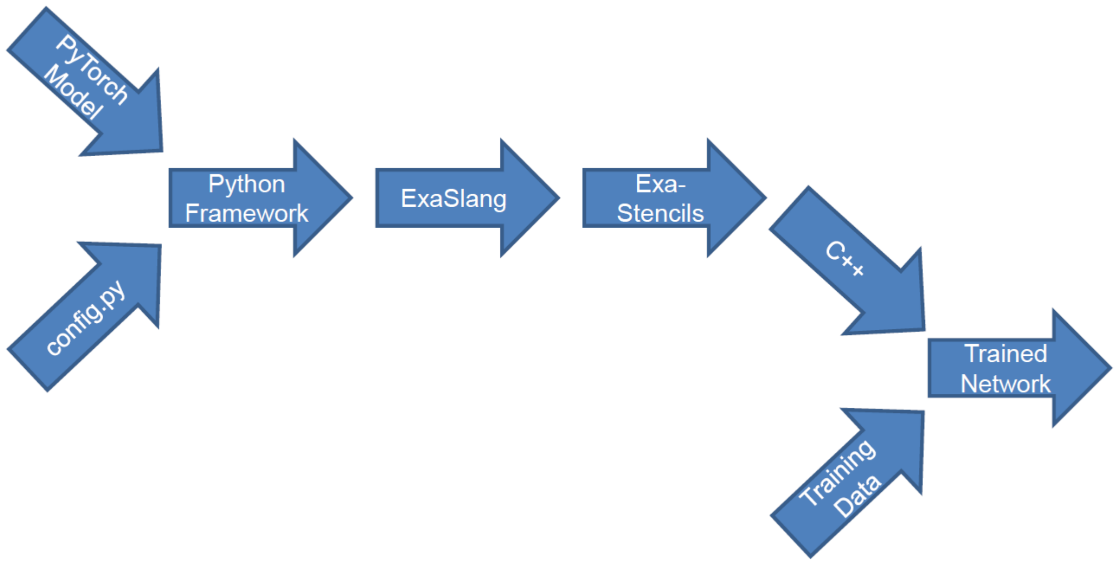


Figure 13: Workflow of the code generation toolchain used in this thesis, starting with a PyTorch model and config file (left) and ending with a trained NN (right).

The code generation toolchain starts with a U-Net implemented as a PyTorch model and a config Python file. Both have to be provided by the user. Those files contain the U-Net architecture (PyTorch file) and the training configuration (config file). Thus all necessary information for the training is provided. The Python framework generates a layer 4 training code for the given U-Net and configuration. Afterward, the toolchain uses ExaStencils for generating C++ code from the layer 4 code. Thus, the pipeline consists of two code generation frameworks. Finally, the executable resulting from the C++ code uses the training data to perform the training. The config file specifies the location and attributes of the training data.

5 Implementation

This section introduces important parts of the Python framework implementation mentioned in section 4. The framework's title is: "Generation of Neural Networks deployed as ExaSlang Specifications". The idea on which the framework is based comes from the GHODDESS framework, which was introduced in [9]. GNNESS reuses plenty of code from GHODDESS, which is a code generation framework for generating layer 4 code to simulate oceans for real-world scenarios by discretizing the 2D shallow-water equations. The user has to provide the simulation hyperparameters in a config file. Finally, GHODDESS uses ExaStencils to generate C++ source code from the layer 4 specification. The resulting executable then computes the ocean simulating using a given grid.
GNNESS also generates layer 4 code given a config file. Section 6.2.2 provides details about the config file. In addition, GNNESS receives a Python file containing a PyTorch model. After the C++ source code generation, the training data is the input for the executable. The following section discusses the implementation details of GNNESS.
The whole training process consists of the following steps:

1. Generate layer 4 code from the config and PyTorch files using GNNESS
2. Generate C++ code from the layer 4 code using ExaStencils
3. Compile C++ code using a general-purpose compiler
4. Execute the executable using the provided training images, i.e., train the NN
5. Optionally: export the trained NN as an ONNX file

The first step is described in detail below since it contains the main work of this thesis, i.e., the implementation of GNNESS. Steps 2 - 4 mainly call ExaStencils, the Compiler, and the generated executable. The generation process of the layer 4 code using GNNESS consists of the following steps:

1. Initialize global variables
2. Initialize fields
3. Build AST
4. Print knowledge file
5. Print layer 4 code

The following subsections explain the implementation of those steps in detail using code snippets. However, the subsections skip some parts of the code or simplify them to make the presented code more understandable and focus on the essentials. The complete code is in the repository at `https://i10git.cs.fau.de/exastencils/thesis_kemmler_samuel`. The line numbering in the following code snippets does not correspond to the actual line numbering in the repository. The line numbering purpose is to easily reference parts of the code in its description. As before, the explanations refer to code lines using brackets. The code snippet descriptions will be above the corresponding codes in the following subsections.

5.1 Variable Initialization

As described in subsection 3.2.1, the layer 4 code offers the possibility to define global variables. The first step in the GNNESS code is to assemble all global variables that the training code needs. GNNESS extracts this information from the given U-Net architecture. The global variables are collected using a Python dictionary. Those variables are mainly matrix data structures for the weights and biases of the (transposed) convolutional layers, matrix data structures for their gradients, and scalar variables like the loss. The complete source code is in the previously mentioned repository in the file "src/gnness/globalVariableStorage.py".
Every global variable of the resulting layer 4 code is represented in GNNESS using an object. Therefore, three classes are defined (3 - 29). The first class is GlobalVariable (3 - 6) and serves as the

base class for all global variable classes. It has a name and data type. There are two different types of global layer 4 variables used in this thesis: scalar and matrix. Therefore, two additional classes are defined: GlobalScalarVariable (9 - 13) and GlobalMatrixVariable (16 - 29). Both inherit from the GlobalVariable base class. The GlobalScalarVariable class additionally stores the information if it is constant or not (12) and its initial expression (13). The GlobalMatrixVariable class stores the 2D matrix elements (19) and the corresponding initial expression using the matrix elements (21 - 26). The constructor receives the elements as a NumPy array. Finally, the GlobalMatrixVariable class defines a ___call___ function that returns a string containing the layer 4 code for accessing the matrix element (i, j) (28 - 29). The ___call___ function is used later by the AST building process (see subsection 5.3) to add matrix accesses to the AST.

```
1  # globalVariableStorage.py
2
3  class GlobalVariable(object):
4    def __init__(self, name, data_type):
5      self.name = name
6      self.data_type = data_type
7
8
9  class GlobalScalarVariable(GlobalVariable):
10   def __init__(self, name, data_type, is_constant, init_expr):
11     super().__init__(name, data_type)
12     self.is_constant = is_constant
13     self.init_expr = init_expr
14
15
16 class GlobalMatrixVariable(GlobalVariable):
17   def __init__(self, name, data_type, size, elements=None):
18     super().__init__(name, data_type)
19     self.elements = elements
20     self.size = size
21     if self.elements is None:
22       self.init_expr = None
23     else:
24       self.init_expr = "{"
25       # fill "init_expr" string with values of "elements"
26       # ...
27
28   def __call__(self, i, j):
29     return self.name + "[" + str(i) + "][" + str(j) + "]"
```

Listing 10: First part of the globalVariableStorage.py code snippet.

Next, the dictionary for the variables is defined and initialized (30).

```
30 variables = {}
```

Listing 11: Second part of the globalVariableStorage.py code snippet.

For adding variables to the dictionary, the add_variable function adds the given variable object to the dictionary using its name as the key (31 - 35). If the caller adds two variables with the same name, the function prints a warning.

```
31 def add_variable(var):
32   global variables
33   if var.name in variables:
34     print("WARN:", "Adding", var, "but␣it␣already␣exists")
35   variables[var.name] = var
```

Listing 12: Third part of the globalVariableStorage.py code snippet.

Next, the init_global_vars function is defined (36 - 86). Its task is to add all global variables used in the layer 4 code to the dictionary. First, the scalar variables like the loss value and the learning rate are added (37 - 39). Second, four Python variables are defined and initialized to store the total number of weights and biases of all (transposed) convolutional layers (42 - 45). Then, a loop iterates over all layers of the given U-Net (47). The corresponding counters increase accordingly if a layer is either a 2D convolution or a 2D transposed convolution (48 - 64). Finally, the core defines four NumPy arrays for the trainable parameters of the (transposed) convolutional layers using the previously calculated sizes (66 - 69).

```
36 def init_global_vars(config: ConfigMaster):
37   add_variable(GlobalScalarVariable("loss", "Real", False, 0.0))
38   # add further scalar variables for example the learning rate
39   # ...
40
41   # count number of trainable parameters
42   n_conv_weights = 0
43   n_conv_biases = 0
44   n_transposed_conv_weights = 0
45   n_transposed_conv_biases = 0
46
47   for layer in config.layers:
48     if isinstance(layer, Conv2d):
49       n_conv_weights += (
50         layer.kernel_size[0]
51         * layer.kernel_size[1]
52         * layer.in_channels
53         * layer.out_channels
54       )
55       n_conv_biases += layer.out_channels
56
57     if isinstance(layer, ConvTranspose2d):
58       n_transposed_conv_weights += (
59         layer.kernel_size[0]
60         * layer.kernel_size[1]
61         * layer.out_channels
62         * layer.in_channels
63       )
64       n_transposed_conv_biases += layer.out_channels
65
66   conv_weights = np.zeros((n_conv_weights, 1))
67   conv_biases = np.zeros((n_conv_biases, 1))
68   transposed_conv_weights = np.zeros((n_transposed_conv_weights, 1))
69   transposed_conv_biases = np.zeros((n_transposed_conv_biases, 1))
```

Listing 13: Fourth part of the globalVariableStorage.py code snippet.

The remaining part of the init_global_vars function traverses the U-Net layers again (70 - 77). If the current layer is either a convolution or transposed convolution, the code fills the previously created NumPy arrays with the weights and biases of the current layer (71 - 77). This step allows the comparison of the PyTorch training results with those of the GET since the initial weights and biases are the same (given that "torch.manual_seed(n)" is used in the PyTorch model to get reproducible pseudo-random numbers). However, the user can deactivate this functionality since it may become costly if the U-Net has many trainable parameters. Finally, the code adds four matrix variables to the dictionary containing the array values (80 - 83). Finally, four more matrix variables of the same size are added (with 0 as initial values) for the gradients of the loss w.r.t. the trainable parameters (85 - 86).

```
70   for layer in config.layers:
71     if isinstance(layer, Conv2d):
72       # fill conv_weights and conv_biases with values of layer
73       # ...
74
75     if isinstance(layer, ConvTranspose2d):
76       # fill transposed_conv_weights and transposed_conv_biases with values of
           layer
77       # ...
78
79   # add data structures for trainable parameters
80   add_variable(GlobalMatrixVariable("conv_weights", "Real", (n_conv_weights, 1),
       conv_weights))
81   add_variable(GlobalMatrixVariable("conv_biases", "Real", (n_conv_biases, 1),
       conv_biases))
82   add_variable(GlobalMatrixVariable("transposed_conv_weights", "Real", (
       n_transposed_conv_weights, 1), transposed_conv_weights))
83   add_variable(GlobalMatrixVariable("transposed_conv_biases", "Real", (
       n_transposed_conv_biases, 1), transposed_conv_biases))
84
```

```
85    # add the same data structures as in "add data structures for trainable
          parameters" again for the gradients, but this time with zero values
86    # ...
```

Listing 14: Fifth part of the globalVariableStorage.py code snippet.

These code snippets conclude the first part of the layer 4 code generation. The following subsection discusses the field initialization.

5.2 Field Initialization

As described in subsection 3.2.1, the layer 4 code offers the possibility to define fields on different levels. The following code snippets add all fields used during the training process to a dictionary. The fields will serve as data storage for the input, output, and intermediate images between the layers. The entire code is in the file "src/gnness/fieldStorage.py".
The forward and backward passes require fields. The training cannot reuse the fields of the forward pass since they are needed later in the backward pass (see subsection 3.1.2) for the gradient w.r.t. the trainable parameters. As for the global variables, the code adds one object for every field to a dictionary. Therefore, a class called MatrixCellField is defined (3 - 14). This class represents a 2D array that stores a 2D matrix in each cell. For every pixel of a training batch, $b * c$ values have to be stored, where b is the number of images per batch and c is the number of channels per image. Hence, the matrix size for each pixel is (b, n). The MatrixCellField class stores the field's name and the levels on which it is defined (5 - 9). In addition, the class stores the field's layout and the size of the cell matrices (10 - 11). These variables contain all the information needed to define a field in the layer 4 code. Finally, the MatrixCellField class defines a __call__ function (13 - 14), which returns a string containing the layer 4 code for accessing the matrix element (batch, channel) of the current field element. The __call__ function is used later by the AST building process (see subsection 5.3) to add field accesses inside loops to the AST.

```
1   # fieldStorage.py
2
3   class MatrixCellField(object):
4     def __init__(self, name, layout, matrix_size, levels=None):
5       self.name = name
6       if levels is None:
7         self.levels = []
8       else:
9         self.levels = levels
10      self.layout = layout
11      self.matrix_size = matrix_size
12
13    def __call__(self, batch, channel):
14      return self.name + "[" + str(batch) + "][" + str(channel) + "]"
```

Listing 15: First part of the fieldStorage.py code snippet.

Next, the dictionary for the fields is defined and initialized (15).

```
15  fields = {}
```

Listing 16: Second part of the fieldStorage.py code snippet.

The code defines an add_field function (16 - 20). It adds the given field object to the dictionary using its name as the key. If the caller adds two fields with the same name, the function prints a warning.

```
16  def add_field(field):
17    global fields
18    if field.name in fields:
19      print("WARN:", "Adding", field, "but␣it␣already␣exists")
20    fields[field.name] = field
```

Listing 17: Third part of the fieldStorage.py code snippet.

Next, the init_fields function is defined (21 - 26). Its task is to add all fields to the dictionary needed in the training process of the layer 4 code. In the beginning, the code adds two fields

(23 - 24) which serve as storage for the input and reference image. Hence, they exist on the finest level. The code defines a suitable layout for each field at the beginning of the layer 4 code. However, this section does not cover the layout definitions in GNNESS. For more details, see the file "src/printer/layer4.py".

```
21 def init_fields(config: ConfigMaster):
22   field_count = 0
23   add_field(MatrixCellField("ReferenceImage_matrix", "ReferenceImageLayout", (
         config.input_shape[0], config.out_channels), levels=[config.maxLevel]))
24   add_field(MatrixCellField("Image" + str(field_count) + "_matrix", "Image0Layout
         ", (config.input_shape[0], config.input_shape[1]), levels=[config.maxLevel
         ]))
25
26   field_count += 1
```

Listing 18: Fourth part of the fieldStorage.py code snippet.

Finally, a for loop iterates over all layers of the given U-Net (29). The code adds two fields if the layer of the current iteration is a convolution (30 - 32) since the convolution generates a new intermediate image. The first new field will serve as storage for the forward pass output of the corresponding convolution, the second field for the backward pass output. If the current loop iteration layer is a transposed convolution, the code does not add a new field. Instead, the previously created fields are also defined on the level below (36 - 39) since the transposed convolution is nothing else than a restriction to the coarser level. If the current loop iteration layer is a pooling layer, the code does not add a new field as well. Instead, it defines the previously created field on the level above (41 - 44). Again, this is because the pooling is nothing else than a restriction to the next finer level.

```
27   # add fields for forward pass and backward pass
28   current_level = config.maxLevel
29   for layer in config.layers:
30     if isinstance(layer, Conv2d):
31       add_field(MatrixCellField("Image" + str(field_count) + "_matrix", "Image" +
             str(field_count) + "Layout", (config.input_shape[0], layer.
           out_channels), levels=[current_level]))
32       add_field(MatrixCellField("ErrorTensorImage" + str(field_count) + "_matrix"
           , "Image" + str(field_count) + "Layout", (config.input_shape[0], layer.
           out_channels), levels=[current_level],))
33
34       field_count += 1
35
36     if isinstance(layer, ConvTranspose2d):
37       current_level += 1
38       access_scalar_cell_field("Image" + str(field_count - 1) + "_matrix").levels
           .append(current_level)
39       access_scalar_cell_field("ErrorTensorImage" + str(field_count - 1) + "
           _matrix").levels.append(current_level)
40
41     if isinstance(layer, AvgPool2d):
42       current_level -= 1
43       access_scalar_cell_field("Image" + str(field_count - 1) + "_matrix").levels
           .append(current_level)
44       access_scalar_cell_field("ErrorTensorImage" + str(field_count - 1) + "
           _matrix").levels.append(current_level)
```

Listing 19: Fifth part of the fieldStorage.py code snippet.

The previously described codes conclude the second part of the layer 4 code generation. The following section describes the building process of the AST, which is the most crucial part of GNNESS since it assembles the training code into an AST that gets printed into the layer 4 file in the final step.

5.3 AST Building Process

This subsection describes the crucial part of the GNNESS code, the AST building process. As described in subsection 12, the AST serves as an abstract representation of the layer 4 code. After building the AST, GNNESS prints the layer 4 code into a file by iterating through the tree and

printing the code snippets that correspond to the nodes.
The code defines a Node class (3 - 5) which is the base class for all node types. There are two types of nodes: expressions and statements. Hence, the Expression and Statement classes are defined (8 - 15). They both inherit the Node class and serve as base classes for the node types. Since the final GNNESS step traverses the AST and prints the layer 4 code for every node into a file, every node has to define a pretty_print function that returns the corresponding layer 4 code. Expressions are simple (single-line) code snippets, for example, addition or function call. On the other hand, statements are more complex code structures like functions, loops, or conditions.

```
1  # node.py
2
3  class Node:
4    def pretty_print(self):
5      pass
6
7
8  class Expression(Node):
9    def pretty_print(self):
10     pass
11
12
13 class Statement(Node):
14   def pretty_print(self):
15     pass
```

Listing 20: Code snippet from node.py.

One statement type is the Function class (18 - 33). The function class contains a name, the body, a list of parameters, the levels on which it exists, and its return type (20 - 26). Again, the function body lists nodes representing the code inside the function. In the ASTs terminology, these body nodes are the function's children. Finally, the pretty_print function is defined. It returns the layer 4 function skeleton and the results of all pretty_print functions of the body nodes (28 - 33). The print_exa function (29) either calls pretty_print (if p is a node), the name (if p is a field), or directly the string (if p is a string).

```
16 # function.py
17
18 class Function(Statement):
19   def __init__(self, name: str, body: [Statement], parameters=None, levels="",
       return_type="Unit"):
20       self.name = name
21       self.body = body
22       if parameters is None:
23         parameters = []
24       self.parameters = parameters
25       self.levels = levels
26       self.return_type = return_type
27
28     def pretty_print(self):
29       ret = ("Function␣" + self.name + self.levels + "␣(␣" + ",␣".join(print_exa(
           p) for p in self.parameters) + "␣)")
30       if self.return_type != "Unit":
31         ret += "␣:␣" + self.return_type
32       ret += "␣{\n" + "" + ("\n".join(print_exa(s) for s in self.body)) + "\n}\n\
           n"
33       return ret
```

Listing 21: Code snippet from function.py.

Next, the dictionary for the functions is defined and initialized (36). It stores a node object for every function that is part of the layer 4 code.

```
34 # functionCollection.py
35
36 functions = {}
```

Listing 22: First part of the functionCollection.py code snippet.

The code defines the add_function function (37 - 41). It adds the given function object to the dictionary using its name as the key. If the caller adds two function objects with the same name, the function prints a warning. In addition, the get_function function (44 - 47) returns the corresponding node object for a given level if it exists. Otherwise, it is added to the dictionary if add_if_missing=True.

```
37 def add_function(fct: Function, level):
38   global functions
39   if fct.name in functions:
40     print("WARN: ", "Adding", fct.name, "but␣it␣already␣exists")
41   functions[fct.name + str(level)] = fct
42
43
44 def get_function(fct, config, level, add_if_missing=True):
45   if fct.fct_name not in functions and add_if_missing:
46     add_function(fct(config, level), level)
47   return functions[fct.fct_name + str(level)]
```

Listing 23: Second part of the functionCollection.py code snippet.

The following code snippet shows one expression node example (50 - 55). It is the Addition class. It receives a summands list (51) which contains node objects representing the addition's summands. Its pretty_print function returns the layer 4 code for an addition with a variable number of summands(54 - 55).

```
48 # expressions.py
49
50 class Addition(Expression):
51   def __init__(self, summands):
52     self.summands = summands
53
54   def pretty_print(self):
55     return "(␣" + "␣+␣".join(print_exa(s) for s in self.summands) + "␣)"
```

Listing 24: First part of the expressions.py code snippet.

The code snippet below presents another important expression node called InternalFunctionCall (56 - 72). It receives either a function name or object, the function call arguments, and the level it operates on. Finally, the pretty_print function returns the layer 4 code of a function call (67 - 72).

```
56 class InternalFunctionCall(Expression):
57   def __init__(self, fct, level="", args=None):
58     if args is None:
59       args = []
60     if isinstance(fct, str):
61       self.fct = functionCollection.get_function(fct)
62     else:
63       self.fct = fct
64     self.level = level
65     self.args = args
66
67   def pretty_print(self):
68     ret = self.fct.name
69     if len(self.level) > 0:
70       ret += "@" + self.level
71     ret += "␣(␣" + (",␣".join(print_exa(s) for s in self.args)) + "␣)"
72     return ret
```

Listing 25: Second part of the expressions.py code snippet.

The following example presents the Assignment node (75 - 82), which inherits from the Statement node. It receives a left-hand side, a right-hand side, and optionally an operator. Its pretty_print function returns the corresponding layer 4 assignment code (81 - 82).

```
73 # statements.py
74
75 class Assignment(Statement):
76   def __init__(self, lhs, rhs, op="="):
77     self.lhs = lhs
78     self.rhs = rhs
```

```
79      self.op = op
80
81    def pretty_print(self):
82      return print_exa(self.lhs) + "␣" + self.op + "␣" + print_exa(self.rhs)
```

Listing 26: First part of the statements.py code snippet.

Another essential Statement node is the ForLoop (83 - 93). It receives the number of iterations, the body, and optionally a count variable (85 - 90), which increases by one in every loop iteration. Again, the body is a list of nodes representing the code executed in the loop body. Finally, the pretty_print function is defined (92 - 93). It returns the layer 4 code repeat loop skeleton with the results of all pretty_print functions of the body nodes.

```
83  class ForLoop(Statement):
84    def __init__(self, repetitions: int, body: [Statement], count_variable=""):
85      self.repetitions = repetitions
86      self.body = body
87      if count_variable != "":
88        self.count_statement = "count␣" + count_variable
89      else:
90        self.count_statement = ""
91
92    def pretty_print(self):
93      return ("repeat␣" + str(self.repetitions) + "␣times␣" + self.count_statement
          + "␣{\n" + "␣␣" + ("\n␣␣".join(s.pretty_print() for s in self.body)) + "\
          n}")
```

Listing 27: Second part of the statements.py code snippet.

The next code snippet of the AST building process is the Application function node (96 - 124). It represents the main routine of the layer 4 code. Inside its ___init___ function, the AST is built by appending nodes to the self.body list. The first node that the function adds to the body list is the call of the "InitWeights" function (100). For simplicity, most of the nodes that are part of the body list are skipped (102). In addition, the ___init___ function adds a for loop to the application function body (104 - 124). It contains an assignment (108), another for loop (109), and one more assignment (110 - 120). By calling get_function (100), add_function is called (see the definition of get_function (44 - 47)) in case the function dictionary does not yet contain InitWeights. If add_function is called (46), this executes the ___init___ of InitWeights, starting the InitWeights AST building process. This way, the AST is built recursively by calling the ___init___ of the Application node. Finally, every function is inside the function dictionary.

```
94  # application.py
95
96  class Application(Function):
97    fct_name = "Application"
98    def __init__(self, config: ConfigMaster):
99      super().__init__(self.fct_name, [])
100     self.body.append(InternalFunctionCall(functionCollection.get_function(
            InitWeights, config, level=config.maxLevel), level=str(config.maxLevel)))
101
102     # ...
103
104     self.body.append(
105       ForLoop(
106         config.learning_epochs,
107         [
108           Assignment(globalVariableStorage.get_variable("loss"), 0.0),
109           ForLoop(config.num_batches, training_loop_stmts),
110           Assignment(
111             globalVariableStorage.get_variable("loss"),
112             Multiplication(
113             [
114               globalVariableStorage.get_variable("num_pixels"),
115               config.input_shape[0],
116               config.num_batches
117             ]
118             ),
119             "/=",
            ),
            Print(["loss"]),
          ],
        )
      )
```

Listing 28: Code snippet from application.py.

The following code snippet shows the layer 4 code that results from the GNNESS Application class code shown above.

```
Function Application ( ) {
  // further function calls are skipped
  // InternalFunctionCall
  InitWeights@9 ( )
  ...
  // ForLoop
  repeat 5 times {
    // Assignment
    loss = 0.0
    // ForLoop
    repeat 16 times {
      // training_loop_stmts
      ForwardPass@9 ( )
      BackwardPass@9 ( )
      Optimization@9 ( )
    }
    // Assignment
    loss /= ( num_pixels * 8 * 16 )
    // Print
    print ( loss )
  }
}
```

Listing 29: Layer 4 code of the "Application" function.

The next code example builds the AST of a ReLU layer backward pass (see subsection 3.1.4). First, the corresponding field objects are accessed and stored using the access_scalar_cell_field function of the fieldStorage (3 - 8). They are the error tensor that this ReLU layer operates on and the corresponding field from the forward pass. The code for the ReLU layer consists of three nested loops. The outer one iterates over the image pixels, the middle one over the batches, and the inner loop loops over all channels. The code defines two list variables (10 - 11) that it uses to store the AST nodes of the loop body codes. The outer field loop declares a batch variable (13 - 15), which is the loop variable of the batch loop. The middle loop defines a variable called channel (16 - 18), which is the loop variable of the channel loop. The code adds a for loop to the batch loop body (19 - 38). This loop iterates error_tensor_matrix_size[1] times, corresponding to the channel size. After every loop iteration, the channels variable is incremented (36). The code has a condition node inside the for loop. The resulting code checks if the image_field_matrix is <= 0 (24 - 25) for the corresponding batch and channel. If so, it sets the error_tensor_matrix for the corresponding batch and channel to 0 (27 - 32). Finally, the code adds the loop that iterates over the batches to the field loop statements (39 - 45) and adds the field loop to the AST (46 - 48).

```
1  # forward.py
2
3  image_field_matrix = fieldStorage.access_scalar_cell_field(
4    "Image" + str(conv_count) + "_matrix"
5  )
6  error_tensor_matrix = fieldStorage.access_scalar_cell_field(
7    "ErrorTensorImage" + str(conv_count) + "_matrix"
8  )
9
10 field_loop_stmts = []
11 batch_loop_stmts = []
12
13 field_loop_stmts.append(
14   VarDecl("batch", 0, data_type="Int")
15 )
16 batch_loop_stmts.append(
  VarDecl("channel", 0, data_type="Int")
)
batch_loop_stmts.append(
  ForLoop(
    error_tensor_matrix.matrix_size[1],
    [
      Condition(
        image_field_matrix("batch", "channel")
        + "␣<=␣0",
        [
          Assignment(
            error_tensor_matrix(
            "batch", "channel"
            ),
            0.0,
          )
        ],
      )
    ],
    count_variable="channel",
  )
)
field_loop_stmts.append(
  ForLoop(
    config.input_shape[0],
    batch_loop_stmts,
    count_variable="batch",
  )
)
self.body.append(
  FieldLoop(error_tensor_matrix, field_loop_stmts)
)
```

Listing 30: ReLU code snippet from forward.py.

The previous GNNESS code finally results in the following layer 4 code.

```
loop over ErrorTensorImage2_matrix {
  Var batch : Int = 0
  repeat 8 times count batch {
    Var channel : Int = 0
    repeat 1 times count channel {
      if ( Image2_matrix[batch][channel] <= 0 ) {
        ErrorTensorImage2_matrix[batch][channel] = 0.0
      }
    }
  }
}
```

Listing 31: Layer 4 code of the ReLU forward pass.

The final step of GNNESS is printing all dictionaries that have been introduced (among other things) into a layer 4 file.

5.4 Layer 4 Code Printing

After collecting all global variables, fields, and functions in the corresponding dictionaries, the final step is to print them into a layer 4 file, the input for ExaStencils. The process of generating the layer 4 code consists mainly of printing the field declarations (5), the functions (6), and the global variables (7). The following code snippets skip a few minor parts of the layer 4 code for the sake of simplicity.

```
# layer4.py

def print_layer4(config: ConfigMaster):
  with open(config.filename_l4, "w") as f:
    print_field_decls(f)
    print_functions(f)
    print_global_vars(f)
    # print a few more little things
```

```
 9        # ...
```

Listing 32: First part of the layer4.py code snippet.

The print_global_vars function iterates over the variable dictionary and prints the corresponding variable declaration and definition into the given file. The function checks if a variable is either scalar (13) or a matrix (18). If it is scalar, it can be constant (14) or not (16). The variable definition is then printed into the given file using the corresponding variable name and initial expression (15). The code differentiates between matrix variables with and without initial conditions (19 - 22).

```
10  def print_global_vars(f):
11    print("Globals␣{", file=f)
12    for variable in globalVariableStorage.variables.values():
13    if isinstance(variable, globalVariableStorage.GlobalScalarVariable):
14      if variable.is_constant:
15        print("␣␣Val", variable.name, "␣:␣", variable.data_type, "␣=␣", variable.
              init_expr, file=f)
16      else:
17        print("␣␣Var", variable.name, "␣:␣", variable.data_type, "␣=␣", print_exa(
              variable.init_expr), file=f)
18    else:
19      if variable.init_expr is None:
20        print("␣␣Var", variable.name, "␣:␣", "Matrix<" + variable.data_type + ",␣"
              + str(variable.size[1]) + ",␣" + str(variable.size[0]) + ">", file=f)
21      else:
22        print("␣␣Var", variable.name, "␣:␣", "Matrix<" + variable.data_type + ",␣"
              + str(variable.size[1]) + ",␣" + str(variable.size[0]) + ">", "␣=␣",
              variable.init_expr, file=f)
23
24    print("}", file=f)
```

Listing 33: Second part of the layer4.py code snippet.

The function print_functions calls pretty_print for every function in the corresponding dictionary. It writes the results into the given file (25 - 27).

```
25  def print_functions(f):
26    for fct in functionCollection.all_functions():
27      print(fct.pretty_print(), file=f)
```

Listing 34: Third part of the layer4.py code snippet.

Finally, the code prints every field definition into the given file (28 - 30) using the respective name, layout, and levels.

```
28  def print_field_decls(f):
29    for field in fieldStorage.all_fields():
30      print("Field␣" + field.name + "␣<␣global,␣" + field.layout + ",␣None␣>␣" + "@
            (" + (",␣".join(str(level) for level in field.levels)) + ")", file=f)
```

Listing 35: Fourth part of the layer4.py code snippet.

6 Evaluation

The following section reports the evaluation of the GET. Before a performance investigation of the generated code is reasonable, the first step is to ensure that it yields correct training results, i.e., the loss decreases similar to a reference implementation. This check is the topic of subsection 6.1. Subsequently, subsection 6.2 evaluates the usability and user experience. This subsection mainly consists of a comparison between the GET and PyTorch regarding how much effort the user has when training a U-Net, how difficult it is for an experienced PyTorch user to switch to the GET and what limitations are there (at the moment) when training U-Nets using the GET. Afterward, subsection 6.3 presents the performance evaluation. First, the subsection compares the training performance of the first code generation approach (subsection 6.3.3) using OpenMP parallelization with PyTorch for different U-Net sizes and numbers of samples per batch. However, this evaluation reveals performance issues of the generated code regarding the possibility of vectorization (see subsection 6.3.4). Hence, subsection 6.3.5 applies some improvements and evaluates the performance of the second approach. In addition, subsection 6.3.8 presents the performance results of a hybrid MPI-OpenMP parallelization, with the focus on how the code scales across up to 16 CPU nodes. Furthermore, subsection 6.3.9 compares the PyTorch GPU performance with the CPU and GPU performance of the generated codes. Finally, subsection 6.3.10 presents the main memory consumption of PyTorch and the generated codes since this can be a limiting factor for training NNs.

6.1 Correctness

This subsection evaluates the generated code regarding its correctness, which means that the loss outputs of the generated code are compared to PyTorch over the epochs for different training configurations. If the generated code and PyTorch return the same loss values (apart from floating-point inaccuracy), the generated code is correct. This assumption is reasonable since everything happening in the training process finally affects the loss. This subsection performs the correction check for various NN architectures. In addition, it modifies and checks numerous training parameters. However, checking a finite number of scenarios cannot ensure that the GET generates error-free codes for all possible training configurations. PyTorch and the generated code only return the same loss values if they start with the same initial weights and biases. The code sets the random seed of PyTorch to a fixed value (torch.manual_seed(101)) to ensure that the used trainable parameters are indeed the same every time.

6.1.1 Application Scenario

The following subsection describes the image segmentation application scenario used to evaluate the generated code's correctness. The Oxford-IIIT pet data set introduced by Parkhi et al. [21] is used. According to Parkhi et al., this data set is an annotated image collection consisting of different cat and dog breeds. Every image comes with a pixel-level segmentation and a rectangle around the animal's head. However, this thesis only uses the pixel-level segmentation. Since this segmentation assigns every pixel to one of three classes (pet body, background, or pet boundary/accessory), this kind of segmentation is also known as trimap. Figure 14 shows one training image from the Oxford-IIIT pet data set together with the corresponding segmentation mask.

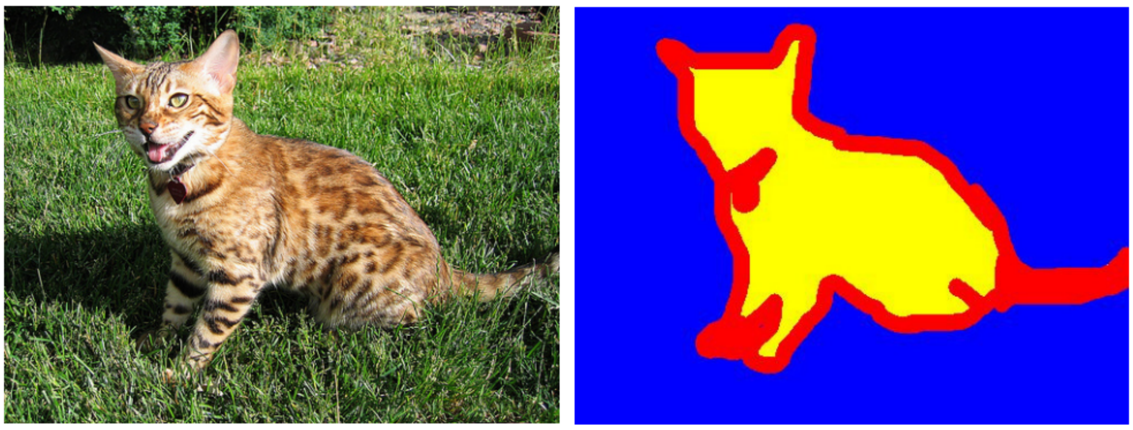


Figure 14: Cat image (left-hand side) and its pixel-level segmentation into three segments (right-hand side) from the Oxford-IIIT pet data set [21].

The correctness evaluation uses only a tiny subset of the Oxford-IIIT pet data set consisting of 128 images to keep the training time within limits. They all belong to the Abyssinian breed. Before the actual training process, the code reshapes the images to a resolution of 512x512. Additionally, it normalizes the images during reading. This way, every pixel value is in the range $[0, 1]$. The evaluation checks the correctness for 32-bit and 64-bit floating-point data structures. The PyTorch loss must not deviate from the generated code beyond floating-point inaccuracy.

6.1.2 Correctness of Different Training Setups

In the beginning, this subsection evaluates different NN architectures regarding their correctness. Furthermore, it tests various training configurations. The architectures are described by listing their layers in the order they are traversed in the forward pass. The architectures consist of four different layer types: the 2D convolution with kernel size 3x3, 0 paddings and stride 1 (conv), the ReLU, the 2D average pooling with kernel size 2x2 and stride 2x2 (pooling), and the 2D transposed convolution with kernel size 2x2 and stride 2x2 (convTransposed). The following list contains all NN architectures that the thesis evaluates regarding their correctness.

- Very simple two layered U-Net: conv → ReLU → pooling → conv → ReLU → convTransposed → conv. There is a skip connection with channel size one connecting the first ReLU's output and the third convolution's input.
- Simple convolutional NN with different channel sizes: conv → conv.
- Simple two-layered U-Net: conv → ReLU → pooling → conv → ReLU → conv → ReLU → convTransposed → conv. There is a skip connection with channel size three connecting the first ReLU's output and the fourth convolution's input.
- U-Net from the publication by Ronneberger et al. [23] (see Figure 2) with varying channel sizes.

The thesis verified that the generated code produces correct results for the following training configurations.

- Input shapes (channels per images, pixels in x direction, pixels in y direction): (1, 4, 4), (3, 4, 4), (3, 512, 512)
- Optimizers: SGD, ADAM (for different learning rates)
- Batch sizes: 8, 64
- Loss function: pixel-wise cross-entropy
- Learning epochs: 2, 5, 50

- OpenMP parallelization (2 - 72 threads) for both code generation approaches, hybrid MPI-OpenMP parallelization (4 and 16 MPI processes, each with 20 OpenMP threads) for the first code generation approach, and CUDA parallelization for the first code generation approach. The further course of this subsection provides details on these code generation approaches.

6.2 Usability and User Experience

An essential aspect that a developer must consider in addition to pure performance is the usability/user experience. The following subsections first give a short overview of how training a NN using PyTorch looks like from a user perspective, then how the user can perform the same training using the GET. Afterward, a subsection compares and discusses both approaches and reports how difficult it is for an (experienced) PyTorch user is to switch. Finally, a subsection discusses the limitations of the GET regarding flexibility and possibilities.

6.2.1 Training a NN Using PyTorch

The following Python code snippet is a minimal example of how to train a NN using PyTorch. For simplicity, the code snippet skips some parts. The code explanation is underneath.

```
 1 import torch as t
 2 import torch.nn as nn
 3
 4
 5 class UNet(nn.Module):
 6   def __init__(self):
 7     super(UNet, self).__init__()
 8     self.conv0 = nn.Conv2d(in_channels=3, out_channels=1, kernel_size=3, stride
         =1,padding=1, bias=True, padding_mode="zeros")
 9     # TODO: add more layers just as above
10
11   def forward(self, x):
12     out = self.conv0(x)
13     # TODO: add data flow for the remaining layers
14     return out
15
16 num_epochs = 5
17 batch_size = 8
18 num_batches = 16
19
20
21 if __name__ == "__main__":
22   model = UNet()
23   loss_function = nn.CrossEntropyLoss()
24   optimizer = t.optim.Adam(model.parameters(), lr=0.001)
25
26   for i in range(num_epochs):
27     average_loss = 0.0
28
29     for b in range(num_batches):
30       # TODO: implement get_next_batch()
31       input, reference_output = get_next_batch()
32       output = model(input)
33       optimizer.zero_grad()
34       loss_value = loss_function(output, reference_output)
35       average_loss += float(loss_value)
36       loss_value.backward()
37       optimizer.step()
38
39     average_loss /= num_batches
40     print(average_loss)
```

Listing 36: Minimal example of how to train a NN using PyTorch.

The code consists of two parts. The first part defines a UNet class inherited from the torch.nn.Module class (5 - 15). It consists of two functions. The first is the __init__ function (6 - 9) used for the initialization of the NN layers, hence the architecture description. One torch object is created and added to the variables for every layer. Those can be, for example, torch.nn.Conv2d, torch.nn.ReLU,

torch.nn.AvgPool2d, etc. The name of the second function is forward (11 - 14). It implements the order in which a training batch traverses the NN in the forward pass. The backward pass does not have to be defined since PyTorch can derive it from the forward pass.
The second part of the code is the main function (21 - 40). Here, an instance of the previously defined UNet class is created (22). In addition, the function defines and initializes a loss function object and the desired optimizer (23 - 24). The training process itself uses two nested for loops. The outer one (26) loops over the number of training epochs, the inner loop iterates over the batches (29). For every batch, the code fetches the input and corresponding reference data. Afterward, the code executes the four training steps described in subsection 3.1.2. First, the forward pass is performed (32). Second, the loss function computes the loss value using the reference image and the result of the forward pass (34). The loss value is added to the average_loss of the whole batch to compute the average loss (35). Third, the code performs the backward pass using the initial error tensor returned by the loss function (36). Finally, the optimizer changes the model's parameters (37). After each epoch, the averaged loss is printed (40). This information is helpful for the user regarding the training progress. Often, the training computes the validation loss after each epoch. That is the loss for a subset of data that the training does not use. The validation loss helps to estimate the NN's performance for samples it has not seen before. However, this is skipped in the code snippet above since it is minimal.
If the Python file pytorch_model.py contains the code above, the user can start the training using the following command.

```
python3 pytorch_model.py
```

Listing 37: Staring the PyTorch training.

The following subsection describes what this training process looks like with the GET.

6.2.2 Training a NN Using the GET

For training a NN, the user has to provide two types of information. First, the NN architecture must be defined, which includes the information which layers the NN consists of and in which order the training batch traverses the layers in the forward pass. When using PyTorch, this information is provided by defining a class (inherited from torch.nn.Module) with an __init__ function describing the NN architecture and a forward function defining the data flow of a batch in the forward pass. Second, the user must define the training process and its parameters like the loss function, optimizer, and the number of training epochs. The previous PyTorch example implements this inside the main function. The PyTorch approach of defining the NN architecture as a class (inheriting from torch.nn.Module) with an __init__ and forward function is very intuitive and convenient. Therefore, the NN architecture definition in the GET works the same. The user defines the NN architecture as a PyTorch model and passes the corresponding file to the GET as input. The training process information (optimizer, loss, number of epochs, batch size) has to be specified by the user in two Python classes called Config and ConfigMaster. The following code snippet shows parts of those configuration files.

```python
class Config(ConfigMaster):
  def __init__(self, pytorch_specification):
    self.training_data_path = "../training_data"
    self.batch_size = 8
    self.learning_rate = 0.001
    self.learning_epochs = 5
    self.loss_function = "CrossEntropyLoss"
    self.optimizer = "ADAM"
    self.export_model_as_onnx = False

  # further variables have been omitted for reasons of clarity
  # ...


class ConfigMaster:
  def __init__(self, pytorch_specification):
  self.use_dbl_precision = False
  self.use_weights_from_model = True

```

```
    # optimization parameters
    self.omp_enabled = True
    self.omp_num_threads = 4
    self.activate_optimization_flags = False

    # further variables have been omitted for reasons of clarity
    # ...
```

Listing 38: Configuration file for the GET.

Variables for all standard training parameters are defined in the ___init___ functions of the configuration files. The user changes the training parameters of the generated code by changing the configuration files. Finally, the user can call the GET using the PyTorch model and the configuration file.

```
python3 unetMain.py pytorch_model.py config.py
```

Listing 39: Staring the GET.

Here, unetMain.py is the main GNNESS file, pytorch_model.py contains the previously mentioned class describing the NN architecture, and the config.py contains the Config class. Since the Config class inherits from ConfigMaster, the user only has to pass the Config class to GNNESS.

6.2.3 Comparison, Limitations, and Discussion

The following subsection compares training a NN with PyTorch to using the GET from a user perspective. In addition, it mentions the most significant limitations. Finally, it discusses how big the hurdle of switching to the GET is for experienced PyTorch users.
Downloading GNNESS and ExaStencils using "git clone" is simple, explained in the README, and should not cause significant complications on a Linux machine. Implementing the NN architecture does not differ between PyTorch and the GET because the user can use the same PyTorch model in both cases. The user experience is, therefore, the same. It is different for setting up the training process and parameters like the optimizer or number of epochs. Here, the GET user does not have to write the training process manually as with PyTorch but sets variables of the configuration files with appropriate values.
On the one hand, specifying the training parameters in a configuration file improves the user experience by reducing the coding effort, saving time, and eliminating potential error sources. If the GET supports the desired training configuration, and the user can set all parameters accordingly, the GET provides a more user-friendly way of training NNs than PyTorch. The user experience has similarities with Keras [4]. However, the disadvantage is the significantly limited flexibility of the training process. There are endless combinations and configurations of how the user can implement the training process. For example, the order of training data loading, the transformations that the code should use on the data during the training, or the stopping criterion.
Furthermore, there are plenty of training parameters the user can modify in PyTorch. For example, the user can choose 13 different optimizers. However, the GET currently supports only two (popular) optimizers, SGD and Adam. In PyTorch, countless optional parameters, for example, weight decay, momentum, and damping, can be specified for each optimizer. Defining those parameters is not supported in the GET. The user can select only the learning rate as it is the most crucial parameter.
In summary, the additional effort for standard training without unique configurations is limited. Thus, the user experience and usability are similar to PyTorch, if not better, since only the NN class code has to be written by the user. Instead of writing the training code, the user only sets variables. However, if a non-standard training process is required, the optimization step should use other optimizers than SGD and Adam, or the user wants to change non-standard parameters, the GET is not yet the best choice. There are further limitations. Currently, it is only possible to train a NN with given training data. A train-test-split to compute the validation loss is not yet supported. In addition, the user has to install non-standard libraries, for example, ImageMagick [28]. While this is not an issue on the local machine, it can cause problems on a high-performance cluster without root rights since this can force the user to install libraries from the source.
Furthermore, the GET does not support data augmentation (like rotation or cropping) at the moment. Finally, the GET is limited to very few layer types (convolution, ReLU, average pooling, and

transposed convolution) and does not support special layer configurations. For an experienced PyTorch user, switching to the GET is possible without any complications since the NN architecture definition remains the same, and specifying the training process is even simpler.

6.3 Performance

This subsection reports the performance measurements for a U-Net architecture similar to the one introduced by Ronneberger et al. [23]. The subsection presents measurements for varying channel sizes to see the performance behavior for different numbers of trainable parameters. Figure 15 illustrates the architecture. In contrast to the original publication, the channel sizes depend on an integer n, which allows reporting performance results for the U-Net architecture of Ronneberger et al., but with different numbers of trainable parameters.

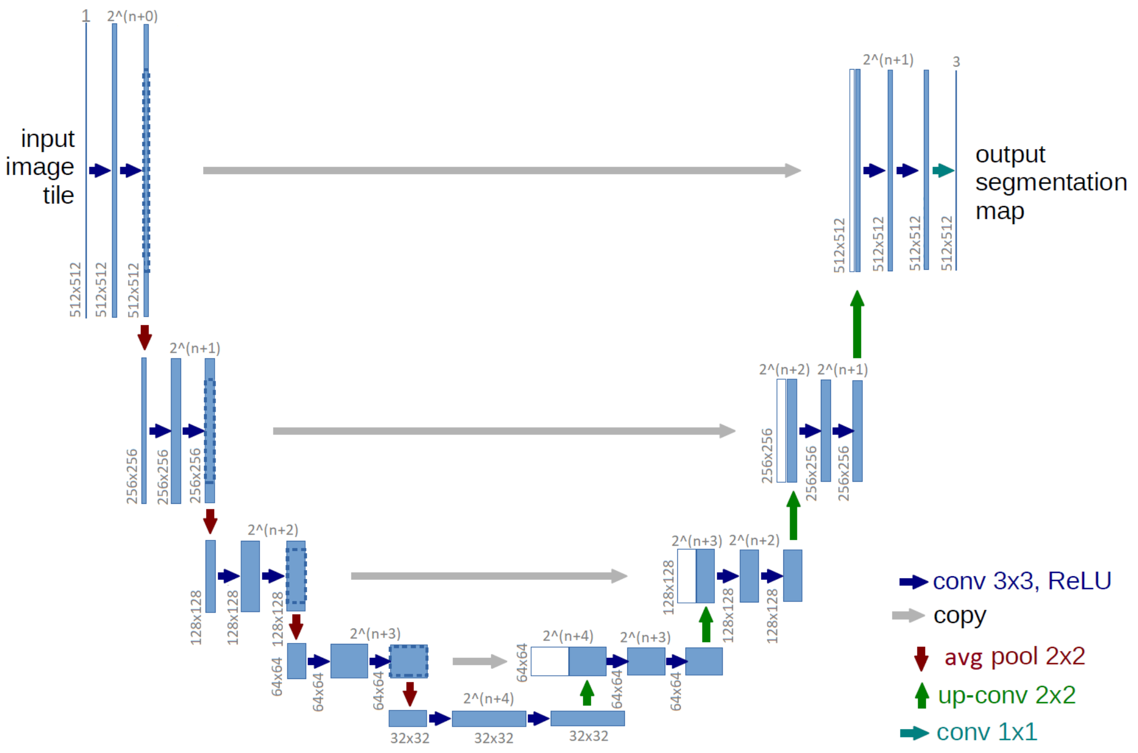


Figure 15: U-Net architecture for the performance evaluations. Blue boxes correspond to multi-channel feature maps. Grey numbers on top of the boxes represent the number of channels, which depends on an integer n. The feature map sizes are denoted on the left or below the boxes. Modification of [23].

The performance results of the further course of this subsection always refer to the architecture presented in Figure 15. They refer to the used U-Net by indicating the n. The following bullet points precisely specify the layers used in Figure 15 by providing the corresponding PyTorch definitions [5]. The copy operation is nothing else than a concatenation in channel direction (see subsection 3.1.1) and therefore is not a separate layer.

- 3x3 convolutional layers (conv 3x3): torch.nn.Conv2d(in_channels=x, out_channels=y, kernel_size=3, stride=1, padding=1, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)
- 1x1 convolutional layers (conv 1x1): torch.nn.Conv2d(in_channels=x, out_channels=y, kernel_size=1, stride=1, padding=1, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)

- 2x2 transposed convolutional layer (up-conv 2x2): torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding=0, groups=1, bias=True, dilation=1, padding_mode='zeros', device=None, dtype=None)
- ReLU activation function (ReLU): torch.nn.ReLU(inplace=False)
- 2x2 average pooling layer (avg pool 2x2): torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True, divisor_override=None)

In the beginning, this subsection reports the hardware and software used for the performance analysis to ensure the results' reproducibility. Afterward, it analyzes the code generation runtimes for the GET, which includes GNNESS, ExaStencils, and the Intel compiler. Furthermore, the section presents the CPU intra-node performance using OpenMP parallelization for two different code generation approaches. The section analyzes those approaches and compares them to the corresponding PyTorch performance. In addition, the section provides a detailed performance analysis for the forward pass of two convolutional layers. Furthermore, it examines the generated hybrid MPI-OpenMP parallel codes regarding their scaling across multiple CPU nodes. In addition, there is a performance comparison of the PyTorch GPU implementation and the CPU and GPU versions of the generated code. Finally, it compares the main memory consumption of PyTorch with the generated codes for different U-Net sizes for both GPU and CPU. Unless otherwise specified, the performance measurements use 32-bit floating-point precision.

6.3.1 Technical Setup

The intra-node CPU performance measurements use a state-of-the-art Intel Xeon Ice Lake CPU node from the "test cluster" of the regional compute center in Erlangen. Table 2 contains the Ice Lake node characteristics.

CPU name	Intel(R) Xeon(R) Platinum 8360Y CPU
Launch date	Q2'21
Code name	Ice Lake
Sockets per node	2
Cores per socket	36
Threads per core	2
SIMD	AVX-512
Clock speed	2.2 GHz (fixed for the measurements)
NUMA domains per node	4
Main memory per NUMA domain	64 GByte/s
Main memory bandwidth (per NUMA domain)	$\sim$ 70 GByte/s
Main memory bandwidth (per socket)	$\sim$ 140 GByte/s
Main memory bandwidth (per node)	$\sim$ 280 GByte/s

Table 2: Characteristics of one Intel Xeon Platinum 8360Y node.

The hardware characteristics of the CPU nodes are evaluated using "likwid-topology". Their main memory bandwidth is measured using "likwid-bench -t update_sp_avx512 -w M0:1GB" since updating a single-precision vector using AVX512 instructions is the most suitable benchmark for this application. This returns about 70 GB/s per NUMA domain for the Ice Lake node. Since the Intel Xeon Ice Lake CPU avoids write-allocates [10], this value is the actual bandwidth.
However, only one of these Ice Lake nodes is currently available at the regional compute center in Erlangen. Therefore, the inter-node performance measurements use up to 16 Intel Xeon Broadwell nodes. Table 3 summarizes the characteristics of one node.

CPU name Code name	Intel(R) Xeon(R) CPU E5-2630 v4 Broadwell
Sockets per node Cores per socket Threads per core SIMD	2 10 1 AVX2
Clock speed	2.2 GHz (fixed for the measurements)
NUMA domains per node	2
Main memory per NUMA domain Main memory bandwidth (per socket) Main memory bandwidth (per node)	32 GByte/s $\sim$ 54 GByte/s $\sim$ 108 GByte/s

Table 3: Characteristics of one Intel Xeon E5-2630 v4 node.

In addition, table 4 presents the hardware characteristics of the GPU node used for the CUDA performance measurements. They are evaluated using the command "nvidia-smi -q" and the datasheet [8].

GPU name Code name	NVIDIA A100 GA100
Compute capability GPU memory size GPU memory bandwidth CUDA Cores FP32 Tensor Float 32	8.0 40 GB 1,555 GByte/s 6912 19.5 TFLOP/s 156 TFLOP/s

Table 4: Characteristics of one NVIDIA A100.

The following tables report the software configurations of the different nodes. Table 5 presents the software configuration of the Intel Xeon Ice Lake node, Table 6 reports the characteristics of the Intel Xeon Broadwell node, and Table 7 contains the software information about the GPU node.

Python	3.8.5
PyTorch	1.10.0
OpenJDK	1.8.0_202
Intel C++ compiler	icpc (ICC) 19.0.5.281
Compiler flags	-fno-alias -O3 -std=c++11 -qopenmp -xCORE-AVX512 -I.
ExaStencils optimization flags	opt_vectorize, opt_useAddressPrecalc, opt_conventionalCSE, data_alignFieldPointers

Table 5: Software configuration of the Ice Lake node.

Python	3.7.6
PyTorch	1.10.1
OpenJDK	1.8.0_202
Intel C++ compiler	icpc (ICC) 17.0.5
Compiler flags	-fno-alias -O3 -std=c++11 -qopenmp -xCORE-AVX512 -I.
ExaStencils optimization flags	opt_vectorize, opt_useAddressPrecalc, opt_conventionalCSE, data_alignFieldPointers

Table 6: Software configuration of the Broadwell node.

Python	3.8.5
PyTorch	1.9.0+cu111
OpenJDK	11.0.13
GCC compiler	10.2.0
NVCC compiler	11.2
GCC compiler flags	-O3 -DNDEBUG -std=c++11 -I. -lcuda -lcudart
NVCC compiler flags	-std=c++11 -O3 -DNDEBUG -lineinfo -arch=sm_80 -I.

Table 7: Software configuration of the A100 GPU node.

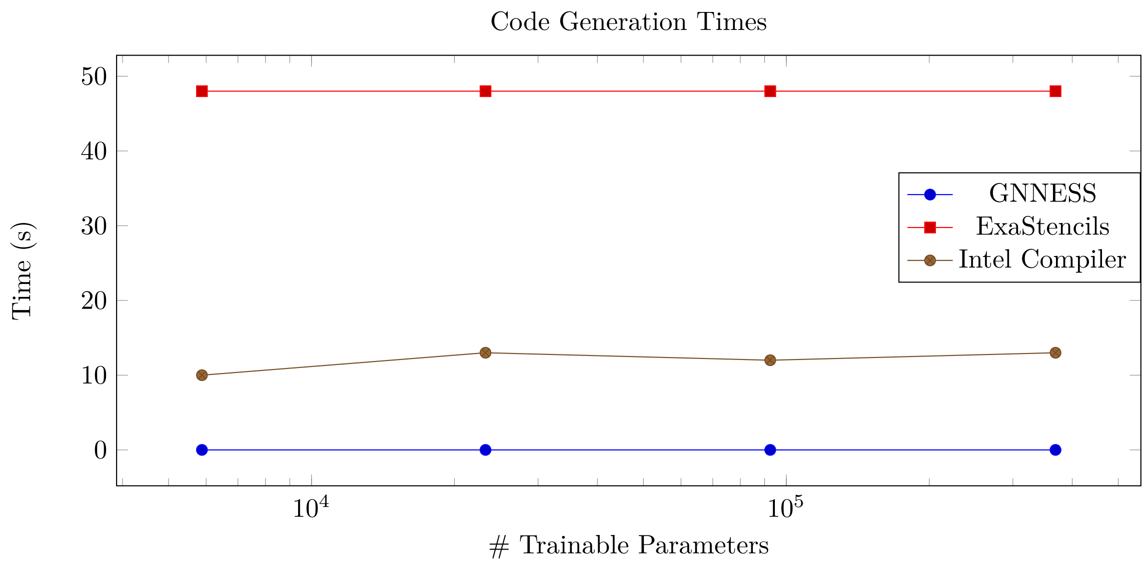
The thesis performs the measurements by submitting job scripts to the slurm batch processing system. For running the PyTorch codes, the job script calls "srun python3 pytorch_model.py". The PyTorch code fixes the thread count using the command "torch.set_num_threads(n)" inside the code. For the generated code, the job script measures the runtime using the command "srun likwid-pin -c 0-$(($i - 1)) ./exastencils" where "i" is the number of threads to be used. The following performance results only consider the pure training: the forward pass, the loss calculation, the backward pass, and the optimization. They use artificially generated training data and do not consider data-related operations like loading or augmentations. All subsequent performance reports ensure that the generated code is correct by comparing the computed loss with PyTorch.

6.3.2 Code Generation

The GET presented in this thesis consists of three different types of code generation:

1. Generation of the layer 4 code from the PyTorch model by GNNESS
2. Generation of the C++ code from the layer 4 specification by ExaStencils
3. Generation of the executable from the C++ code by the (Intel) compiler

The GET offers the possibility of using the same initial weights and biases for training as the PyTorch model instead of randomly initializing them. This feature mainly serves the convenient debugging, as it allows to compare the training losses of the generated code with those of PyTorch. However, this feature has been disabled for the following graph, as it massively increases the code generation time of GNNESS for large networks. In the following measurements, ExaStencils uses the optimization flags described in subsection 6.3.1. The Makefile uses eight threads to execute multiple recipes simultaneously (make -j 8). The following graph shows the code generation times for OpenMP parallel codes for different numbers of trainable parameters. The following subsections analyze U-Nets of four different sizes. All of them are variants of the architecture reported in Figure 15. The smallest one has 5,879 trainable parameters (which corresponds to n = 0), the second one has 23,253 trainable parameters (n = 1), the third one has 92,495 trainable parameters (n = 2) and the biggest investigated U-Net consists of 368,955 trainable parameters (n = 3).



The four data points on the x-axis correspond to the U-Net sizes n = 0 to n = 3. The key message of the previous graph is that all three code generation times for OpenMP parallel codes do not increase for an increasing amount of trainable parameters, and they are negligible compared to the training time.

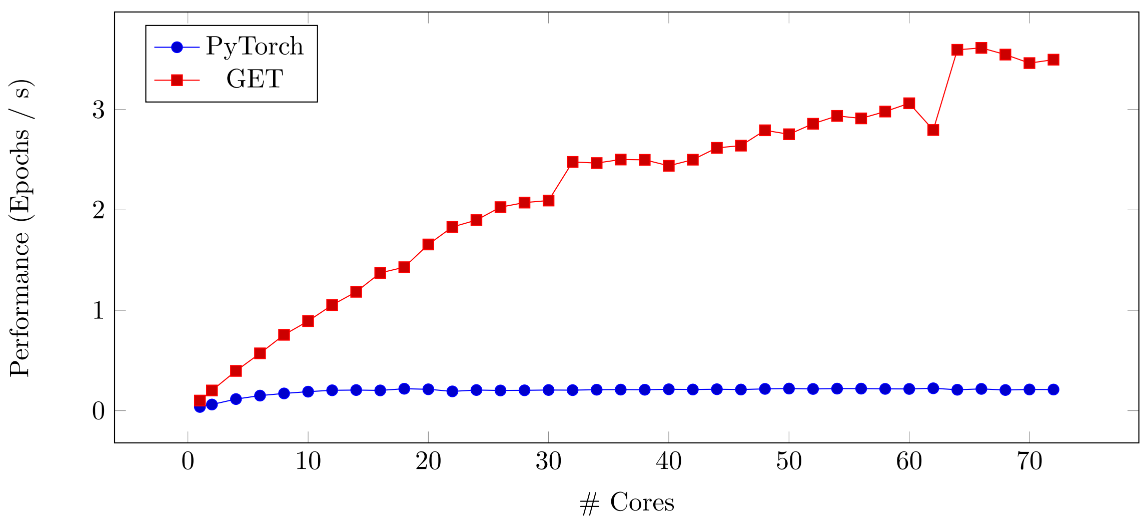
6.3.3 Intra-Node OpenMP Parallelization (First Approach)

The following subsection reports the performance of a first code generation approach. However, this attempt is not optimal. Subsection 6.3.4 describes the flaws of this approach in detail. The further course of the thesis eliminates them, and the performance will be reported again in subsection 6.3.5. For the different U-Net sizes, this subsection reports the performance for different numbers of cores per node up to the entire node. As a performance metric, "epochs per second" is used. The following U-Net performance results again refer to the architecture presented in Figure 15. First, the subsection presents the performance graphs for n = 0 up to n = 3 for both 8 and 64 training samples per batch. After all graphs, the subsection provides an interpretation of the performance results.

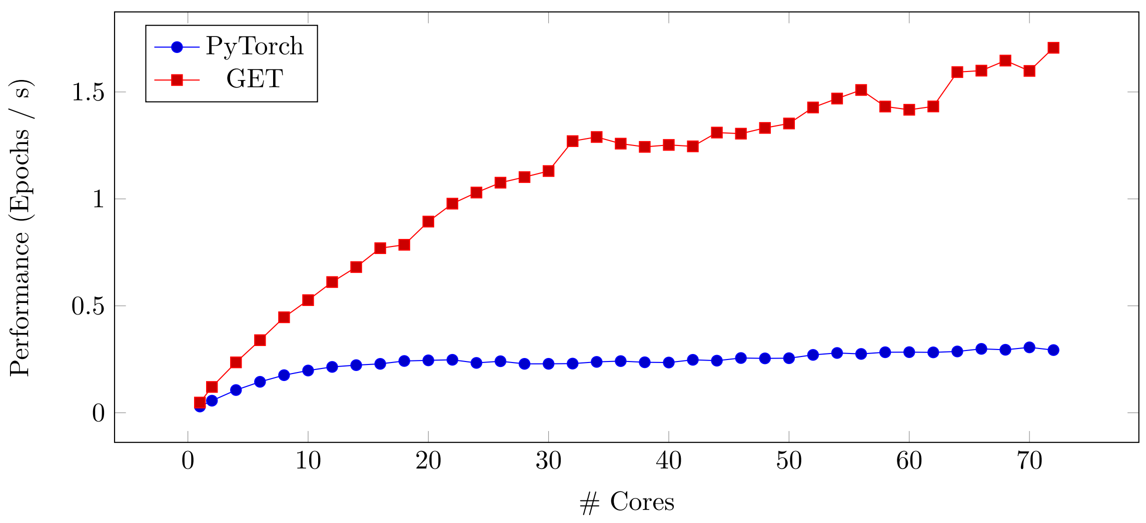
U-Net with 5,879 Trainable Parameters (n = 0)

The performance results for the U-Net with n = 0 (which corresponds to 5,879 trainable parameters) are presented below. The first graph contains the results for eight training samples per batch. The second graph reports the performance for 64 images per batch.

Performance for 8 Samples per Batch



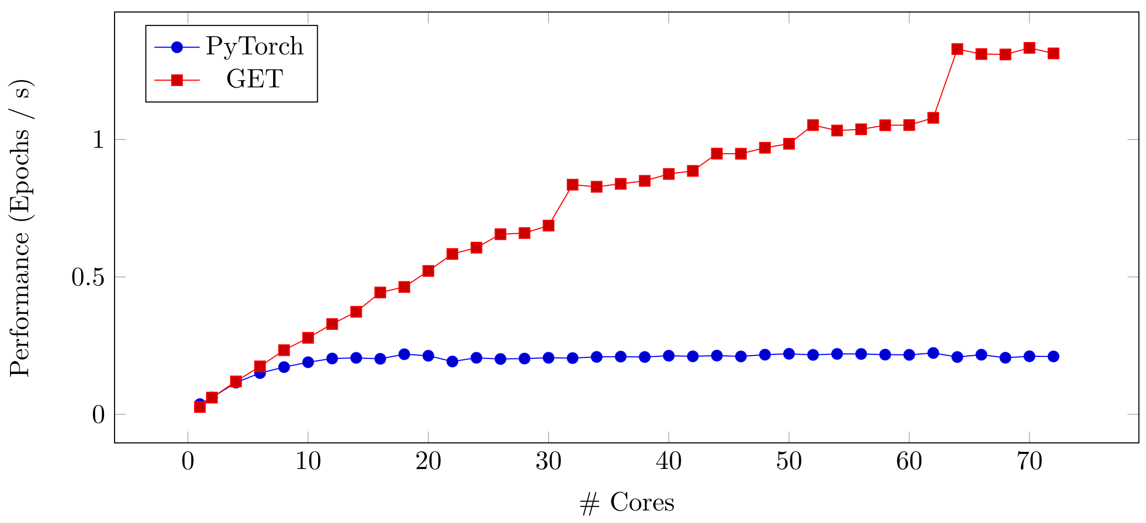
Performance for 64 Samples per Batch



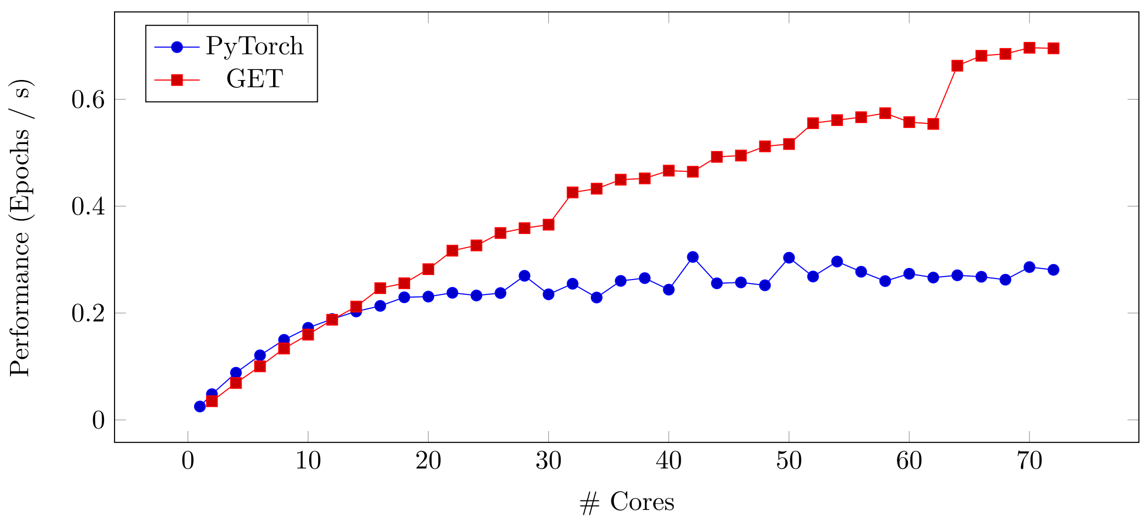
U-Net with 23,253 Trainable Parameters (n = 1)

The performance results for the U-Net with n = 1 (which corresponds to 23,253 trainable parameters) are presented below. The first graph contains the results for eight training samples per batch. The second graph reports the performance for 64 images per batch.

Performance for 8 Samples per Batch

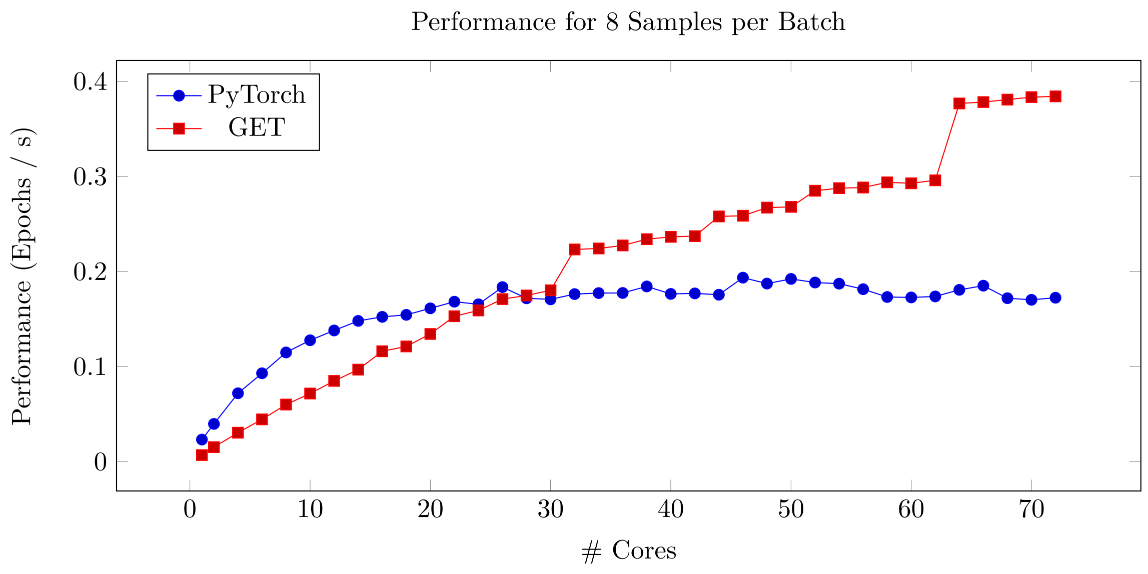


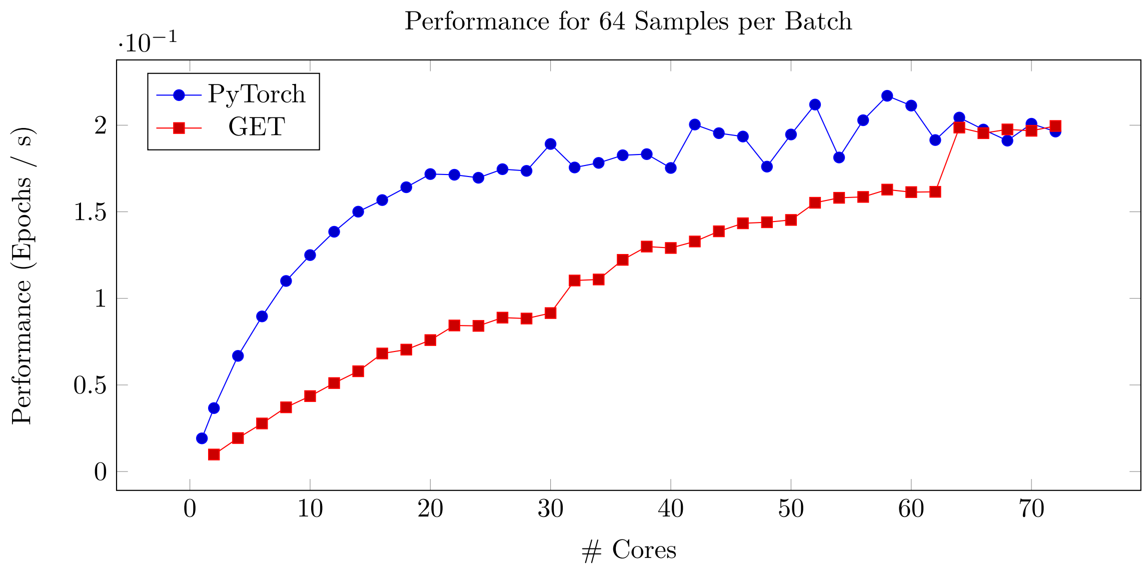
Performance for 64 Samples per Batch



U-Net with 92,495 Trainable Parameters (n = 2)

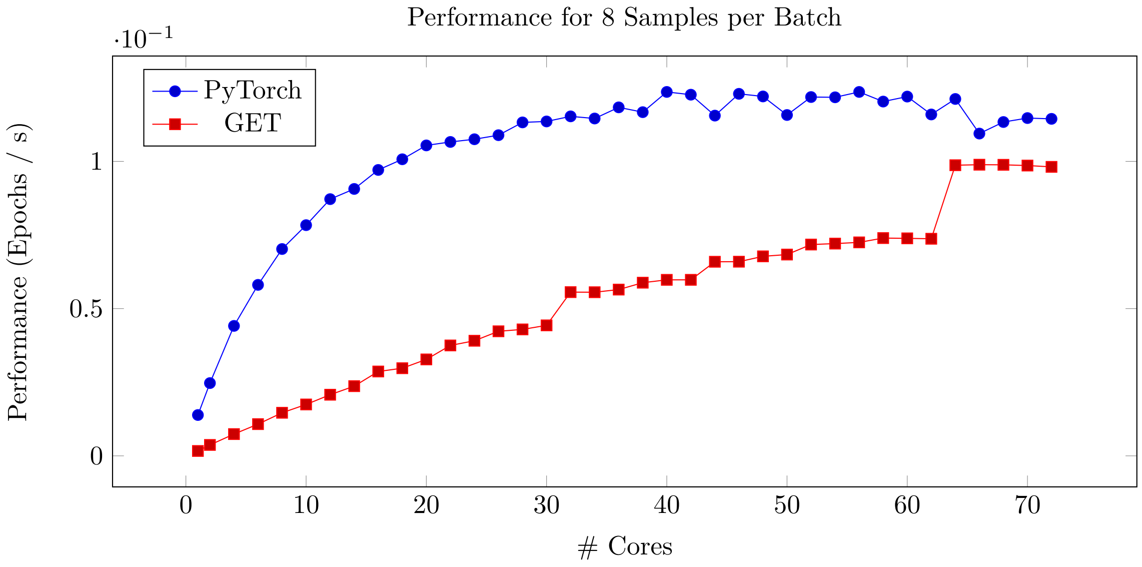
The performance results for the U-Net with n = 2 (which corresponds to 92,495 trainable parameters) are presented below. The first graph contains the results for eight training samples per batch. The second graph reports the performance for 64 images per batch.

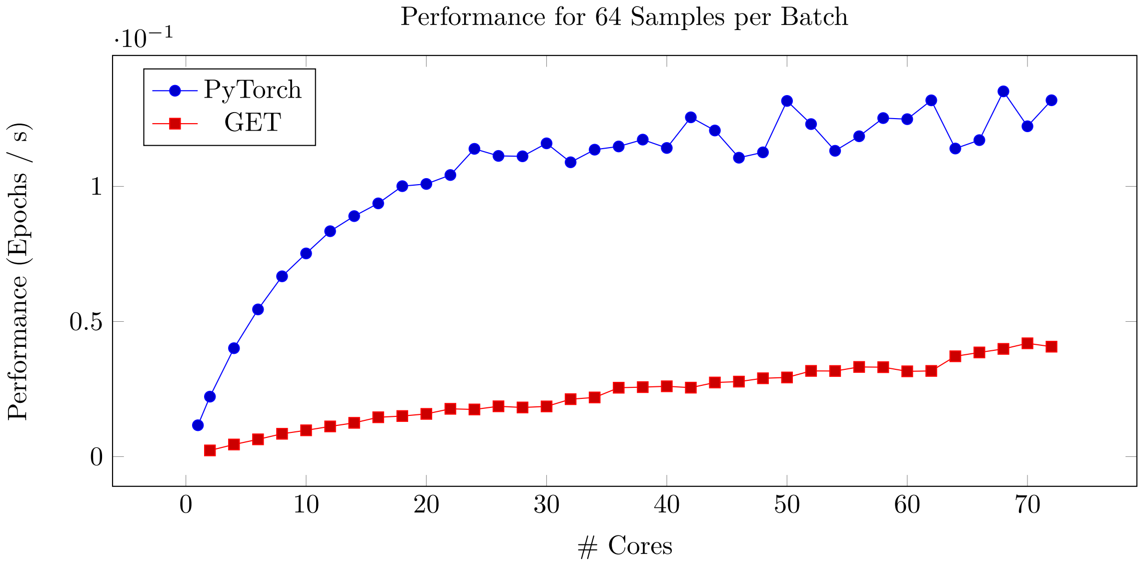




U-Net with 368,955 Trainable Parameters (n = 3)

The performance results for the U-Net with n = 3 (which corresponds to 368,955 trainable parameters) are presented below. The first graph contains the results for eight training samples per batch. The second graph reports the performance for 64 images per batch.





For the smallest U-Net, the GET performance is way above the PyTorch implementation. PyTorch scales poorly across the node. The speedup of the GET compared to PyTorch is roughly 16.6 for eight and 5.8 for 64 samples per batch on the full node. This pattern can be observed similarly for n = 1. Here, the speedup is roughly 6.23 for eight but only 2.47 for 64 samples per batch. The situation changes for n = 2. For a batch size of 64, there is no speedup anymore. For n = 3, this becomes even worse, and the GET achieves far worse performance results than PyTorch. It is noticeable that the performance of the generated code jumps significantly from 31 to 32 and 63 to 64 cores in all graphs. The reason might be the perfect workload balance for 32 and 64 cores since the parallelization is over the outer loop (first pixel dimension). Its size is 512, which is divisible by 32 or 64 without remainder. The performance results show that the GET generated codes are still far away from being optimal in terms of performance. For this reason, the following subsection analyses the performance issues of the generated code and proposes a solution. Then, in subsection 6.3.5, there is again a performance comparison for the improved code generation approach.

6.3.4 Analysis of the Performance Issues of the First Approach

As seen in the previous subsection, the GET performance is worse than the PyTorch implementation for larger U-Net sizes, which implies that the generated code's performance is not optimal. The thesis investigates the single-core runtimes of the different NN layers to locate and optimize the dominating code sections. This information can be found in table 8 for the U-Net with n = 3 and eight training images per batch.

Convolutional layers forward pass	507,8 s
ReLU forward pass	1.6 s
Transposed convolutional layers forward pass	21 s
Pooling layer forward pass	0.4 s
Loss calculation	4.7 s
Initial error tensor calculation	1.7 s
Convolutional layers backward pass	665.9 s
ReLU layers backward pass	2.6 s
Transposed convolutional layers backward pass	5.4 a
Pooling layer backward pass	1.3 s

Table 8: Runtime comparison of the U-Net layers.

Table 8 shows that the convolutional layers dominate the runtime. Therefore, the generated code for the convolutional layer forward pass is analyzed and improved below. However, the same principle also applies to the backward pass. The following code snippet shows the generated code for the forward pass of a convolution with eight samples per batch, 64 output, and three input channels. The following code snippet skips the ExaStencils code optimizations for better readability.

```
for (int fragmentIdx = 0; fragmentIdx<1; ++fragmentIdx) {
  #pragma omp parallel for schedule(static) num_threads(4)
  for (int i1 = 0; i1<512; i1 += 1) {
    for (int i0 = 0; i0<512; i0 += 1) {
      int batch = 0;
      for (batch = 0; batch<8; batch = (batch+1)) {
        int channel = 0;
        for (channel = 0; channel<64; channel = (channel+1)) {
          int channel_input = 0;
          int index0 = ((2113568*channel)+(264196*batch)+(514*i1)+i0+515);
          fieldData_Image1_matrix[index0] = 0.0f;
          for (channel_input = 0; channel_input<3; channel_input = (channel_input
              +1)) {
            fieldData_Image1_matrix[index0] = ((conv_weights[((27*channel)+(9*
                channel_input))]*fieldData_Image0_matrix[((2113568*channel_input)
                +(264196*batch)+(514*i1)+i0)])+ ... +fieldData_Image1_matrix[
                index0]);
          }
          fieldData_Image1_matrix[index0] = (conv_biases[channel]+
              fieldData_Image1_matrix[index0]);
        }
      }
    }
  }
}
```

Listing 40: Convolutional layer forward pass of the first code generation approach.

The issue with this code snippet is that it does not access consecutive memory in the innermost loop iterations. Therefore, no vector instructions are used. The following code snippet presents a second approach to solve this problem. It iterates over the pixels in the two innermost loops instead of the outermost loop as in the previous example. This way, consecutive data accesses are performed in the innermost loop, even if the parallelization is still over the domain as above. The entire loop construct is inside a parallel region, and the implicit OpenMP barrier of the "#pragma omp for" is switched off using a nowait, which prevents the code from bad scaling for a large number of threads. The following code snippet shows the result.

```
#pragma omp parallel num_threads(4) private(batch, channel, channel_input)
{
  for (batch = 0; batch<8; batch = (batch+1)) {
    for (channel = 0; channel<64; channel = (channel+1)) {
      for (channel_input = 0; channel_input<3; channel_input = (channel_input+1))
          {
```

```
        if ((channel_input==0)) {
          for (int fragmentIdx = 0; fragmentIdx<1; ++fragmentIdx) {
            #pragma omp for schedule(static) nowait
            for (int i1 = 0; i1<512; i1 += 1) {
              for (int i0 = 0; i0<512; i0 += 1) {
                fieldData_Image1_matrix[((2113568*channel)+(264196*batch)+(514*i1
                    )+i0+515)] = ((conv_weights[((27*channel)+(9*channel_input))
                    ]*fieldData_Image0_matrix[((2113568*channel_input)+(264196*
                    batch)+(514*i1)+i0)])+ ... +conv_biases[channel]);
              }
            }
          }
        } else {
          for (int fragmentIdx = 0; fragmentIdx<1; ++fragmentIdx) {
            #pragma omp for schedule(static) nowait
            for (int i1 = 0; i1<512; i1 += 1) {
              for (int i0 = 0; i0<512; i0 += 1) {
                fieldData_Image1_matrix[((2113568*channel)+(264196*batch)+(514*i1
                    )+i0+515)] = ((conv_weights[((27*channel)+(9*channel_input))
                    ]*fieldData_Image0_matrix[((2113568*channel_input)+(264196*
                    batch)+(514*i1)+i0)])+ ... +fieldData_Image1_matrix
                    [((2113568*channel)+(264196*batch)+(514*i1)+i0+515)]);
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 41: Convolutional layer forward pass of the second code generation approach.

However, generating a code snippet as presented above is not yet supported by ExaStencils, so GNNESS is using "native()" statements in the layer 4 code for the parallel region around the nested loops. In addition, GNNESS changes the loop parallelization from "#pragma omp parallel for ..." to "#pragma omp for ... nowait". Generating hybrid MPI-OpenMP codes for this loop structure is not possible yet.

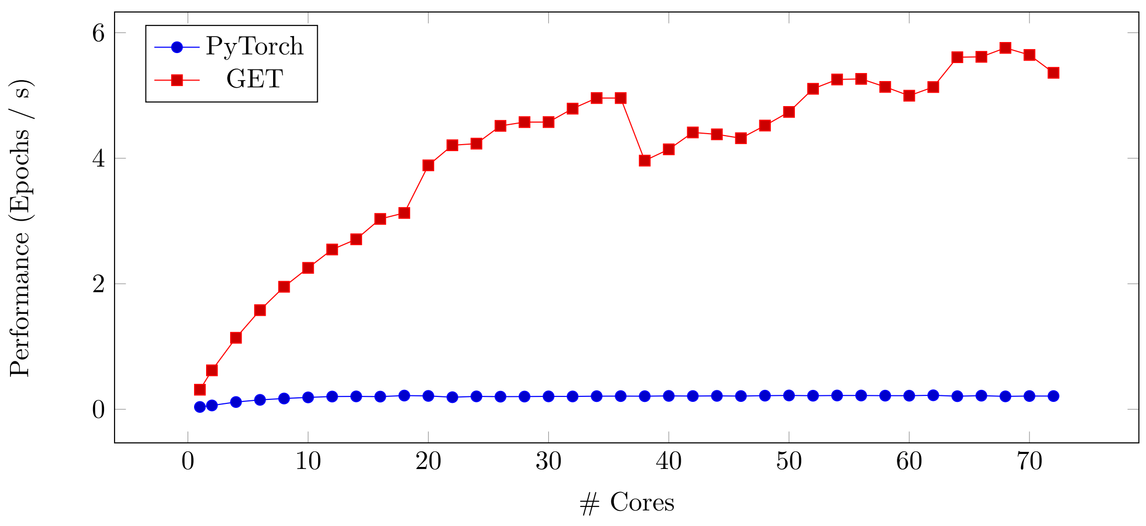
6.3.5 Intra-Node OpenMP Parallelization (Second Approach)

In the following subsection, the performance is reported for the second (improved) code generation approach described in subsection 6.3.4. The performance is reported in the same way and for the same configurations as in subsection 6.3.3.

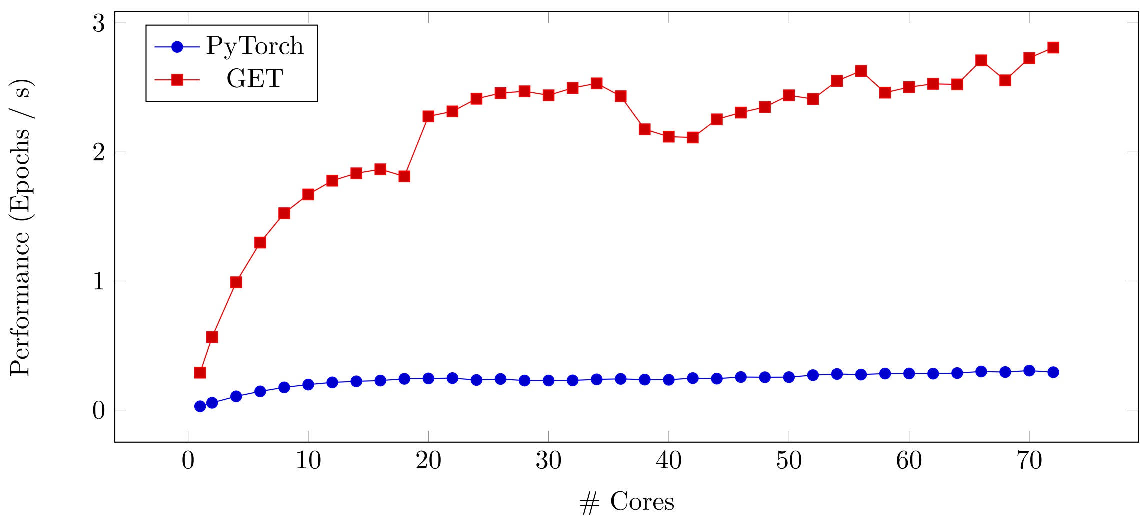
U-Net with 5,879 Trainable Parameters (n = 0)

The performance results for the U-Net with n = 0 (which corresponds to 5,879 trainable parameters) are presented below. The first graph contains the results for eight training samples per batch. The second graph reports the performance for 64 images per batch.

Performance for 8 Samples per Batch

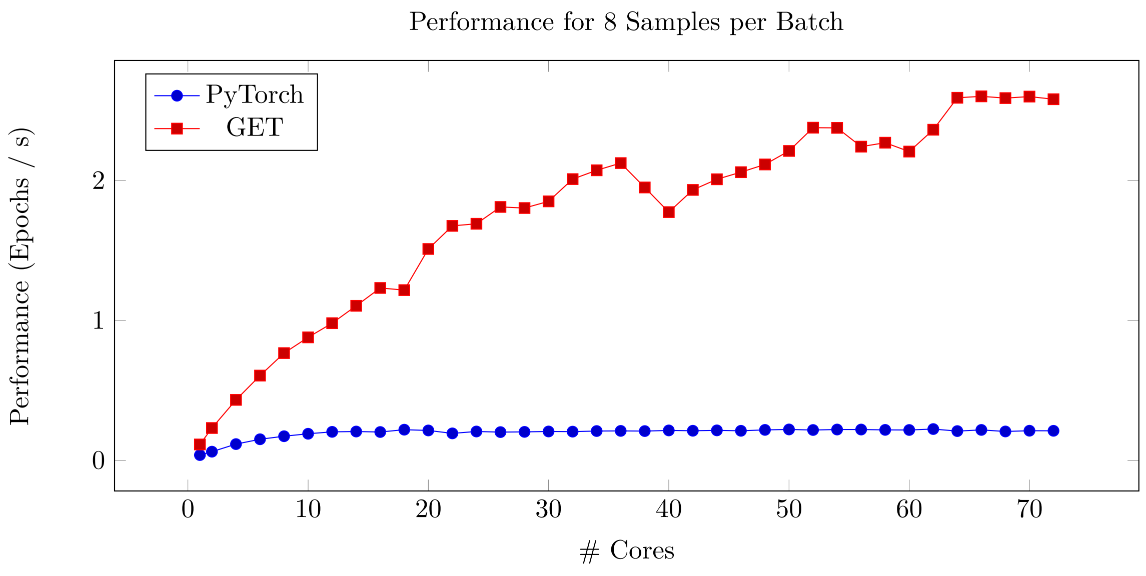


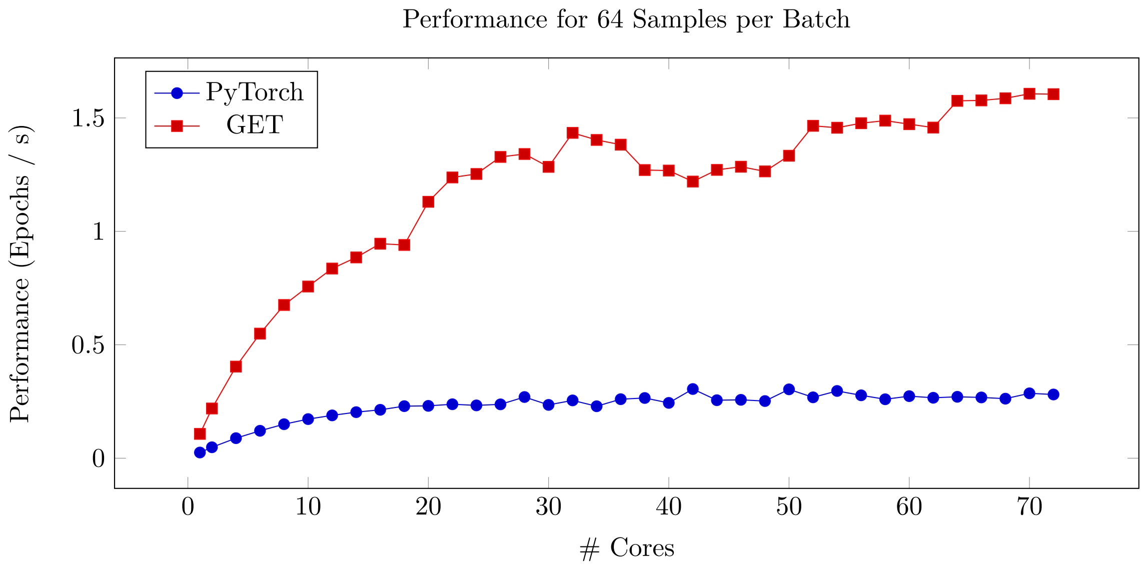
Performance for 64 Samples per Batch



U-Net with 23,253 Trainable Parameters (n = 1)

The performance results for the U-Net with n = 1 (which corresponds to 23,253 trainable parameters) are presented below. The first graph contains the results for eight training samples per batch. The second graph reports the performance for 64 images per batch.

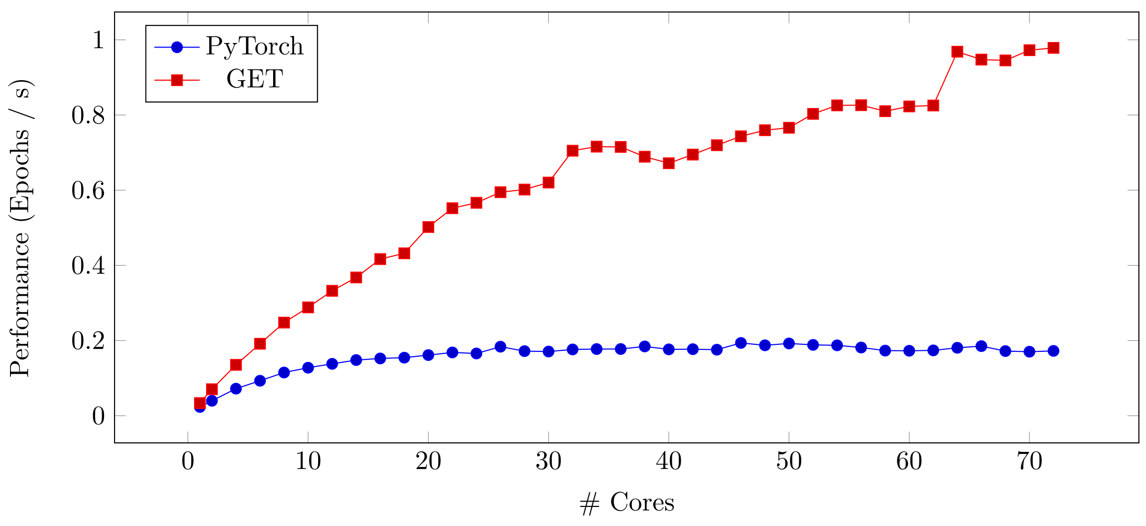




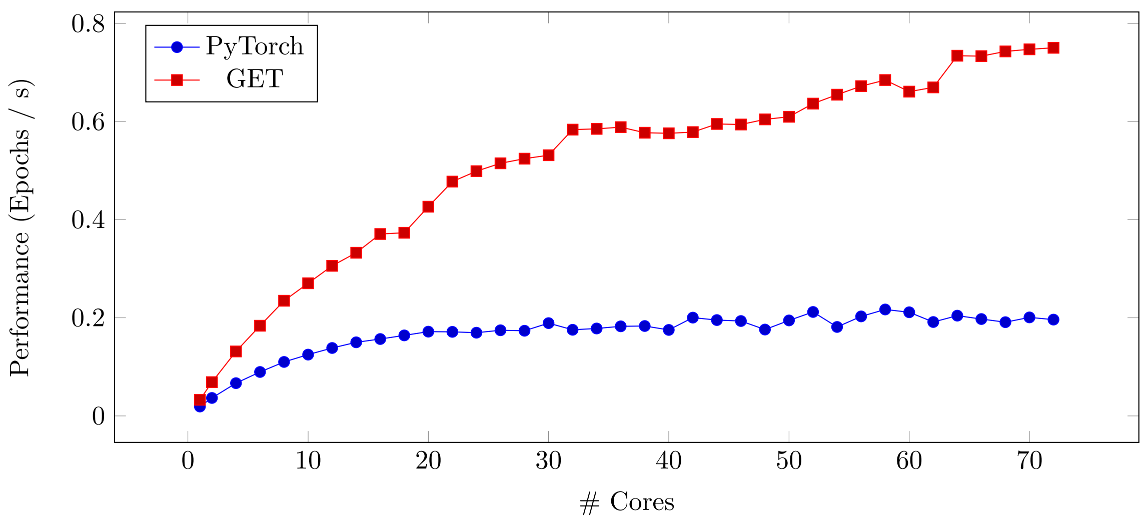
U-Net with 92,495 Trainable Parameters (n = 2)

The performance results for the U-Net with n = 2 (which corresponds to 92,495 trainable parameters) are presented below. The first graph contains the results for eight training samples per batch. The second graph reports the performance for 64 images per batch.

Performance for 8 Samples per Batch

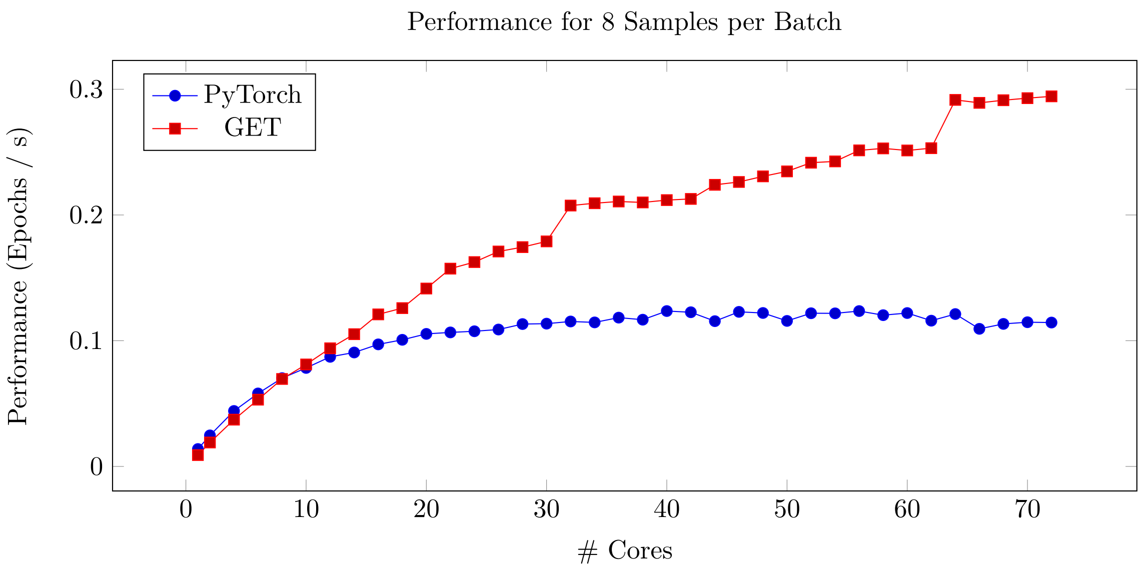


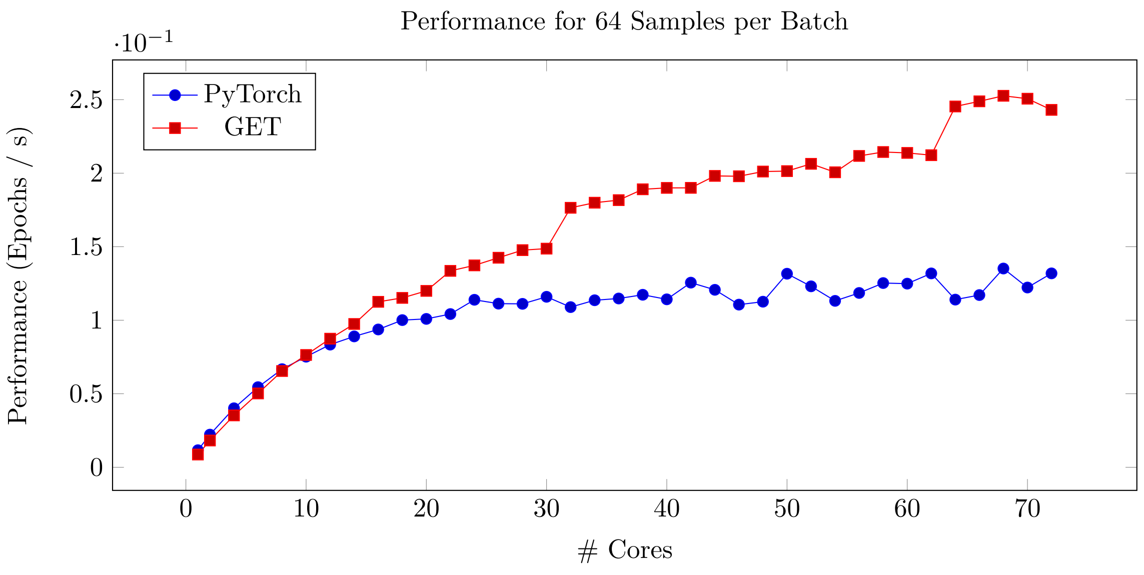
Performance for 64 Samples per Batch



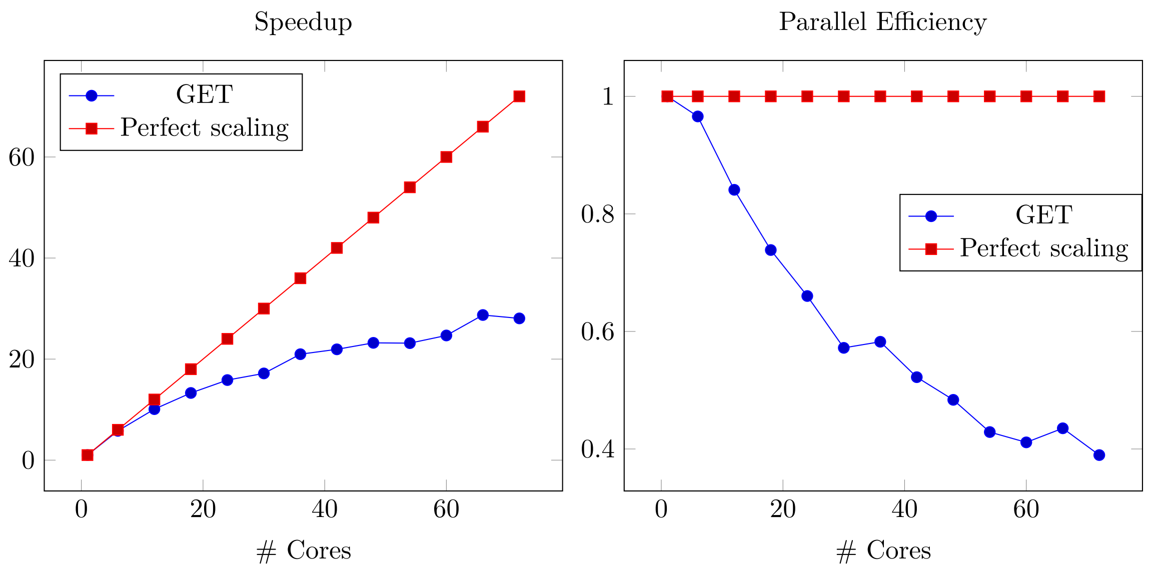
U-Net with 368,955 Trainable Parameters (n = 3)

The performance results for the U-Net with n = 3 (which corresponds to 368,955 trainable parameters) are presented below. The first graph contains the results for eight training samples per batch. The second graph reports the performance for 64 images per batch.





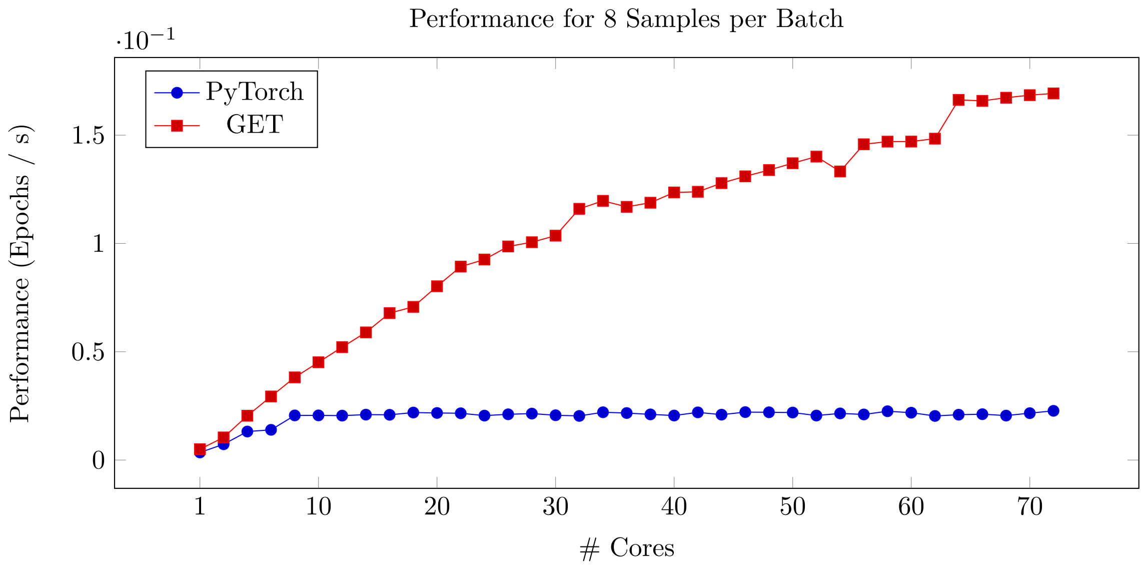
For the small U-Net sizes (n = 0 and n = 1), the improvements do not seem significant compared to the first code generation approach since the old attempt also delivered satisfying results for these small architectures. For n = 2, the difference in performance compared to the first approach is visible for the first time. The generated code still performs better than PyTorch for eight and 64 samples per batch. For n = 2 and n = 3, the performance jumps from 30 to 32, and from 62 to 64 cores is again observable. For n = 3, the difference in performance compared to the first approach is enormous. It is still significantly better than PyTorch for eight and 64 samples per batch. The performance of the generated code for n = 3 and 64 samples per batch has the highest significance since training NNs is typically a computationally intensive task. Therefore, the following two graphs evaluate this configuration's speedup and parallel efficiency.



Both the speedup and the parallel efficiency are satisfying inside the first NUMA domain (up to 18 cores). The speedup is around 13.3, resulting in a parallel efficiency of about 74%. However, the scaling across the NUMA domains needs improvement. For the full node, the speedup is about 28, which is equivalent to a parallel efficiency of less than 40%, which is a considerable drop compared to the first NUMA domain. One factor that influences the scaling negatively is that the reductions of the gradients of the loss w.r.t the trainable parameters were performed on the whole data structure, even if only a fraction was changed. In addition, as can be seen in subsection 6.3.4, the parallelization is applied on the first pixel dimension, i.e., a loop over 512 elements. This unbalanced workload distribution might affect the scaling.

6.3.6 Double Precision

Even if it hardly matters in deep learning, this chapter briefly discusses the performance for double precision. The following graph compares the performance of the generated code for n = 3 and eight samples per batch with PyTorch. PyTorch operates with double-precision by using the command "torch.set_default_tensor_type(t.DoubleTensor)".



The speedup of the GET compared to PyTorch is 7.45, which is significantly higher than for single-precision floating-point operations (2.57) for the same configuration. However, since double precision is not relevant in deep learning, the thesis does not further analyze this topic.

6.3.7 Performance Analysis of the Second Approach

This subsection analyses the forward pass of two convolutional layers with different sizes using the second code generation approach (see subsection 6.3.4) to get insights into how good the performance is so far and if there is potential for optimization. The subsection sets up one Roofline model [20] for each convolution forward pass to get a light speed estimation of the possible performance. According to Hager et al. [11], a single Ice Lake SP core has the following characteristics.

LD/ST throughput per cy:	2 LD + 1 ST
ADD throughput	2 / cy
MUL throughput	2 / cy
FMA throughput	2 / cy
L1-L2 data bus	64 B/cy
L2-L3 data bus	16 + 16 B/cy
L1/L2 per core	48 KiB / 1.25 MiB
LLC	1.5 MiB/core

Table 9: Characteristics of one Ice Lake SP core [11].

In addition, the subsection provides a performance graph for each convolutional layer together with performance measurements, compares them with the light speed estimation, and analyses deviations.

Forward Pass Convolution 0

The following code snippet shows the forward pass of the first convolutional layer for n = 3 and 64 samples per batch. Variable names are changed, and the code computes indices before the accesses for better readability.

```
for (batch = 0; batch<64; batch = (batch+1)) {
  for (channel = 0; channel<8; channel = (channel+1)) {
    for (channel_input = 0; channel_input<3; channel_input = (channel_input+1)) {
      if ((channel_input==0)) {
        // similar to the else case, but with "=" instead of "+="
      } else {
        for (int fragmentIdx = 0; fragmentIdx<1; ++fragmentIdx) {
          for (int i1 = 0; i1<512; i1 += 1) {
            for (int i0 = 0; i0<512; i0 += 1) {
              int index0 = ((27*channel)+(9*channel_input))
              int index1 = (16908544*channel)+(264196*batch)+(514*i1)+i0+515
              int index2 = (16908544*channel_input)+(264196*batch)+(514*i1)+i0
              Image1[4][index1] = (
              (weights[index0]*Image0[4][index2])+
              (weights[index0+1]*Image0[4][index2+514])+
              (weights[index0+2]*Image0[4][index2+1028])+
              (weights[index0+3]*Image0[4][index2+1])+
              (weights[index0+4]*Image0[4][index2+515])+
              (weights[index0+5]*Image0[4][index2+1029])+
              (weights[index0+6]*Image0[4][index2+2])+
              (weights[index0+7]*Image0[4][index2+516])+
              (weights[index0+8]*Image0[4][index2+1030])+
              Image1[4][index1]);
            }
          }
        }
      }
    }
  }
}
```

Listing 42: Convolutional layer forward pass to be analysed.

The following paragraph calculates the light-speed performance estimation using the Roofline model. The main memory bandwidth of one Intel Xeon Ice Lake NUMA domain is roughly 70 GB/s (see subsection 6.3.1). It has to be taken into account that the Intel Xeon Ice Lake avoids write-allocates if the whole cache line is overwritten [10]. The code presented above is executed twice for each epoch since there are two batches. Therefore, Image1 must be written twice per epoch, and Image0 must be loaded twice. The size of Image0 is 512 * 512 (pixels) * 3 (channels per image) * 64 (images per batch) * 4 Bytes ≈ 201 MB. The size of Image1 is 512 * 512 * 8 * 64 * 4 Bytes ≈ 537 MB. Since they are loaded twice this results in (537 + 201) * 2 = 1476 (MB/epoch). Using the bandwidth of 70 GB/s, the upper memory performance limit is 70 GB/s / 1.476 GB/epoch = 47.43 epochs/s. One Ice Lake core can perform 2.2 GHz * 16 (SIMD) * 2 FLOP (FMA) ≈ 70 GFLOPS. Hence, one NUMA domain (18 cores) can perform ≈ 1267 GFLOPS. The computations for one epoch are 512 * 512 * 3 * 8 * 64 (all loops) * 18 (FLOP/kernel) * 2 (batches per epoch) ≈ 14.5 (GFLOP). Hence the upper compute limit is 1267 GFLOPS / 14.5 GFLOP/epoch ≈ 87 epochs/s. So from a Roofline perspective, the code is main memory bound. The following graph reports the actual performance results.

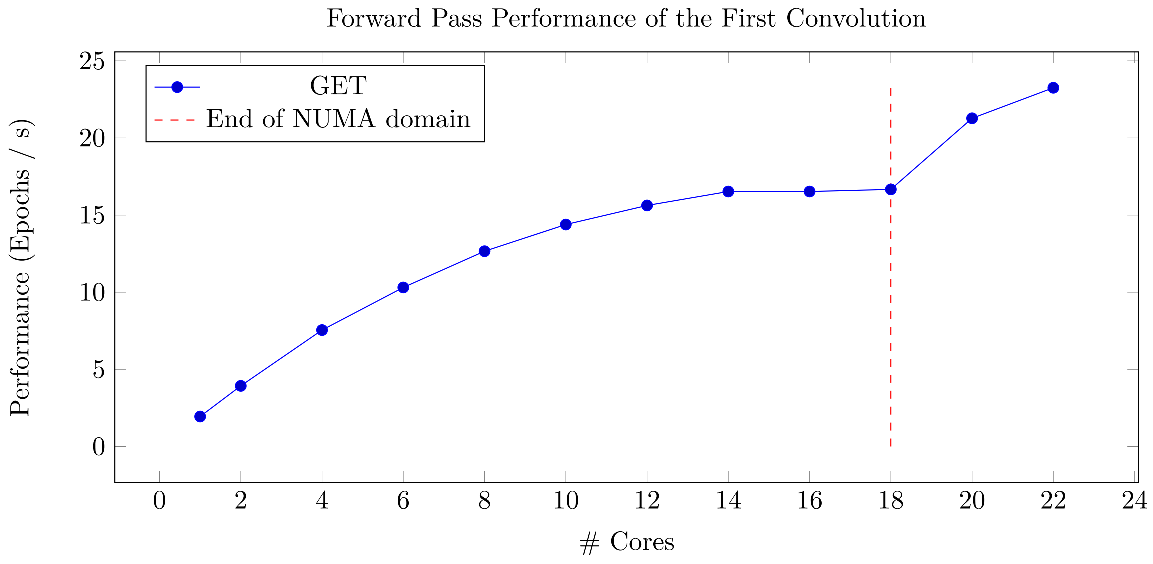


Table 10 reports the measurements for this code snippet for 18 threads and two epochs, i.e., the code snippet is executed four times since there are two batches.

Memory read data volume [GB]	3.8283
Memory write data volume [GB]	3.7365
Memory bandwidth [MB/s]	62562
# FLOP	28.989 GFLOP
L2 bandwidth (max/avg)	16.329/13.968 GBs/s
L3 bandwidth (max/avg)	5.051/4.513 GB/s
Load to store ratio (avg)	7.5731

Table 10: Performance measurements for the first convolution forward pass.

The graph indicates that the code snippet is main memory bound. The number of FLOP is, as expected, twice the number of FLOP per epoch computed above. The measured bandwidth (≈ 63 GB/s) is lower than the maximum bandwidth reported in subsection 6.3.1 (≈ 70 GB/s). An explanation for the difference can be the OpenMP parallelization overhead, different runtimes of the threads, and unbalanced workload distribution, which may lead to many idle threads at some point. Although the code is memory-bound as expected, the performance for 18 threads (16.7 epochs/s) is only one-third of the light speed estimation (52.85 epochs/s). The lower performance comes from much more memory reads and writes than the light speed estimation assumes. The write volume per epoch (1.87 GB) is higher than the estimated volume (537 MB * 2 = 1.074 GB). The read volume per epoch (1.91 GB) is much higher than the estimated volume (201 MB * 2 = 402 MB).

Forward Pass Convolution 1

The forward pass of convolution 1 is very similar to convolution 0. The main difference is that the loop over "channel_input" iterates from 0 to 7 instead of 0 to 2.
The code is executed two times for each epoch since there are two batches. Image1 must be written twice per epoch (since there are two batches), and Image0 must be loaded twice. The size of Image0 is 512 * 512 (pixels) * 8 (channels per image) * 64 (images per batch) * 4 Bytes $\approx$ 537 MB. The size of Image1 is again 512 * 512 * 8 * 64 * 4 Bytes $\approx$ 537 MB. Since they are loaded twice this results in (537 + 537) * 2 = 2148 (MB/epoch). Using the bandwidth of 70 GB/s, this upper memory limit is 70 GB/s / 2.148 GB/epoch = 32.59 epochs/s. The computations for one epoch are 512 * 512 * 8 * 8 * 64 (all of the loops) * 18 (FLOP/kernel) * 2 (batches per epoch) $\approx$ 38.65 (GFLOP). Hence the upper compute limit is 1267 GFLOPS / 38.65 GFLOP/epoch $\approx$ 32.78 epochs/s. So from a Roofline perspective, the memory and compute performance limits are similar.

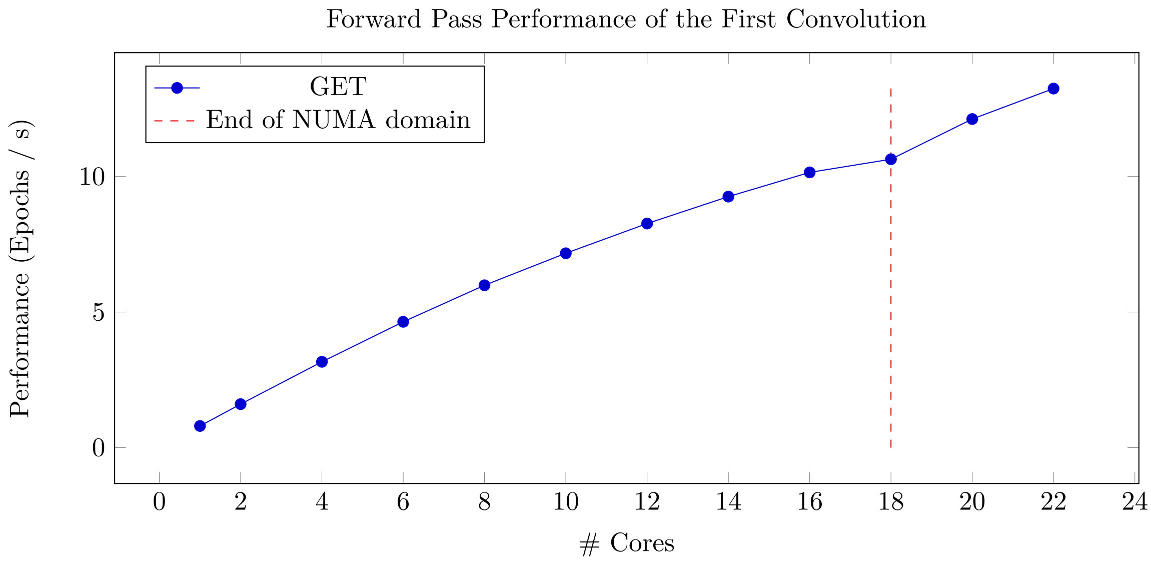


Table 11 reports the measurements for this code snippet for 18 threads and two epochs, i.e., the code snippet is executed four times since there are two batches.

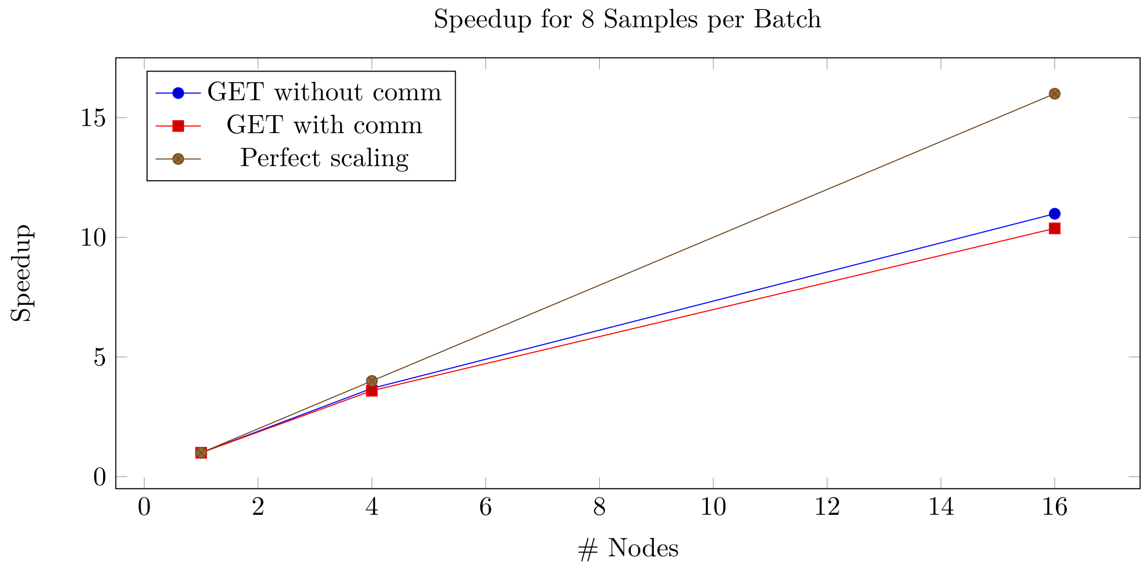
Memory read data volume [GB]	5.8244
Memory write data volume [GB]	3.8550
Memory bandwidth [MB/s]	51185
# FLOP/s	77.306 GFLOP
L2 bandwidth (max/avg)	16.448/16.243 GB/s
L3 bandwidth (max/avg)	3.171/3.124 GB/s
Load to store ratio (avg)	7.7171

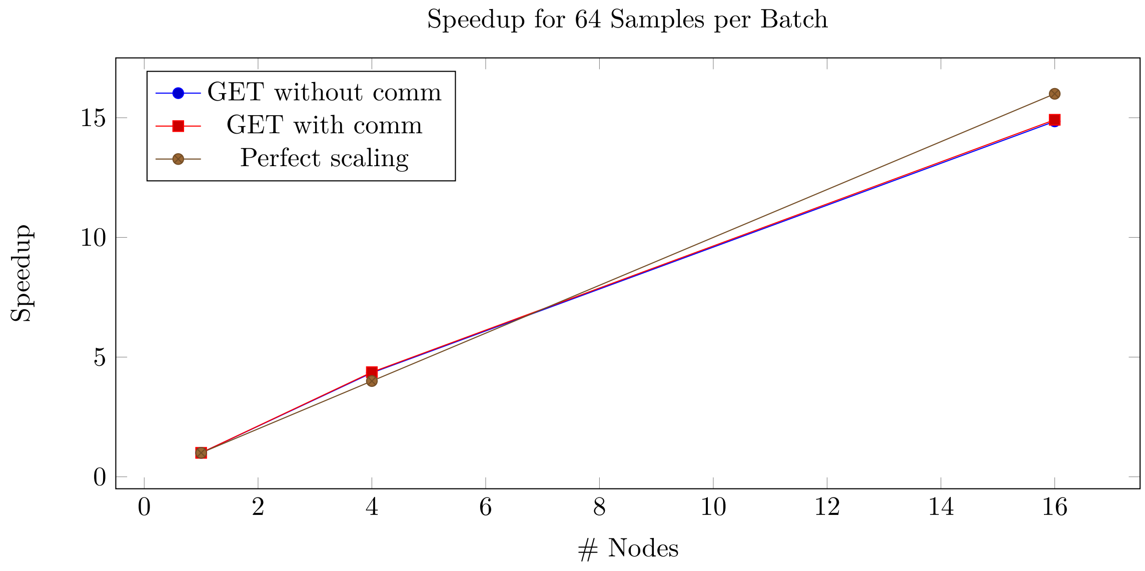
Table 11: Performance measurements for the second convolution forward pass.

The graph indicates that the code snippet is not main-memory bound since there is no saturation at the NUMA domain's end. The number of FLOP is, as expected, twice the number of FLOP per epoch computed above. The measured bandwidth ($\approx$ 52 GB/s) is lower than the maximum bandwidth reported in subsection 6.3.1 ($\approx$ 70 GB/s). The performance for 18 threads (10.6 epochs/s) is only one-third of the light speed estimation (32.59 epochs/s). Again, there are more memory reads and writes than the light speed estimation assumes. The write volume per epoch (1.93 GB) is higher than the estimated volume (537 MB * 2 = 1.074 GB). The read volume per epoch (2.91 GB) is much higher than the estimated volume (537 MB * 2 = 1.074 GB). Since the code seems to be neither main memory-bound nor compute-bound, the thesis assumes that the memory hierarchy limits the performance. Both convolutional layer forward passes only achieve around one-third of the light speed estimation and perform much more main memory traffic than in the light-speed case, which suggests there is potential for future optimizations.

6.3.8 Inter-Node Hybrid MPI-OpenMP Parallelization

This subsection reports the performance of a hybrid MPI-OpenMP implementation. The focus is on how well the generated code scales on multiple nodes. Since the second code generation approach (see subsection 6.3.3) does not yet support MPI parallelization, the following evaluations use the first approach (see subsection 6.3.5). Since only one Intel Xeon Ice Lake node is available, these measurements use Intel Xeon Broadwell nodes. Although this results in worse single-node performance, this is tolerable since this subsection is primarily about scaling.
The following measurements were performed using "likwid-mpirun -n $nodes -pin N:0-19 ./exastencils" in an interactive job. So 20 OpenMP threads are used per node, and the MPI processes parallelize between the nodes. The following graphs show the speedup for 2, 4, and 16 nodes for the U-Net with n = 3 and eight and 64 samples per batch. The following graphs report the performance for two communication approaches: communicate the boundary pixels between the MPI processes after each layer in the forward and backward pass (with comm) or do not communicate them (without comm).

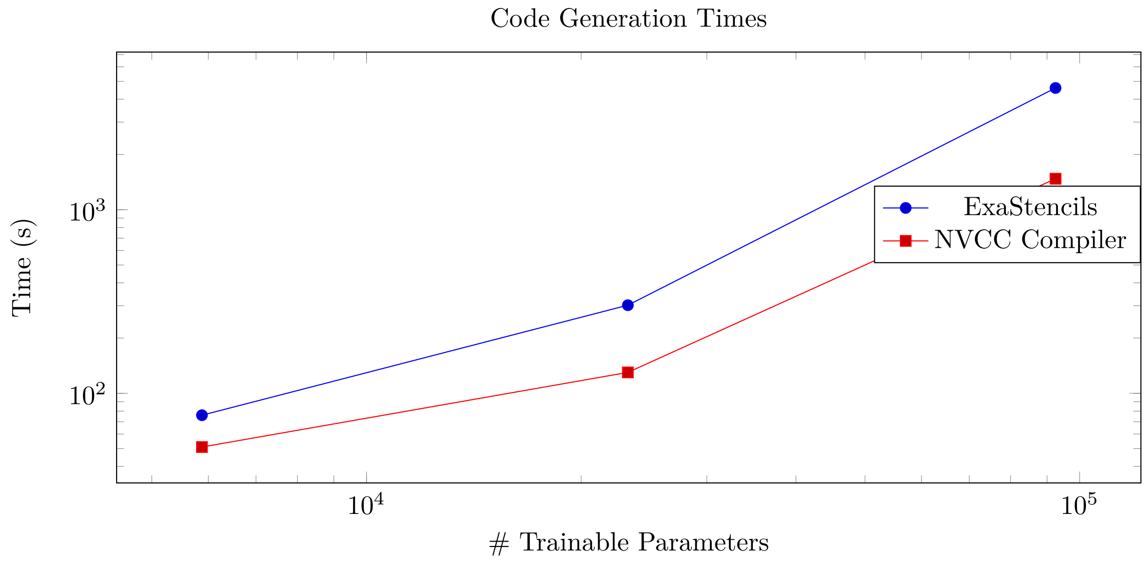




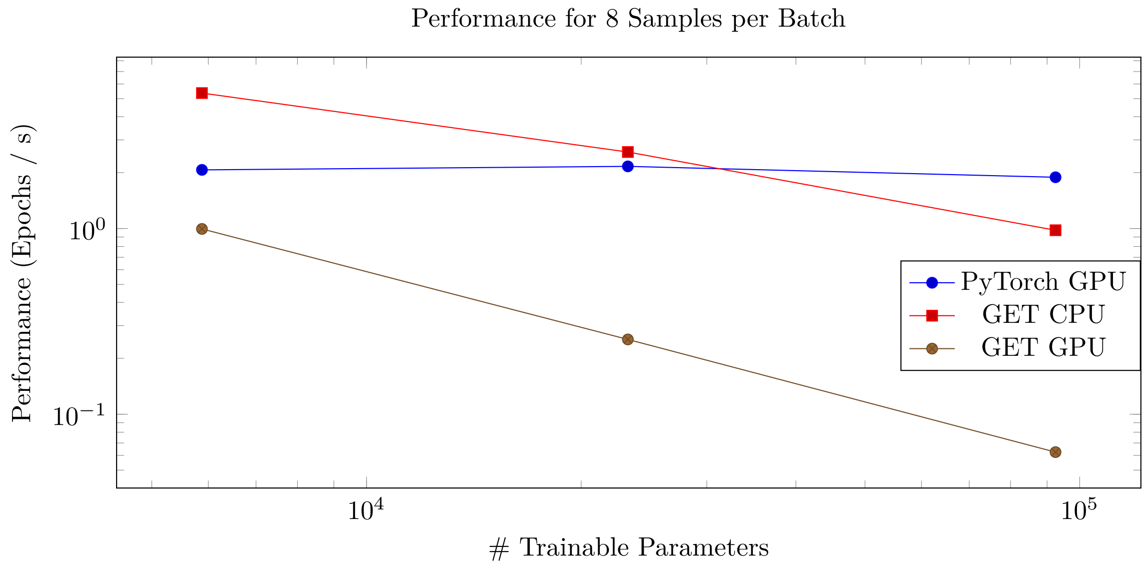
It turns out that the versions with and without communication of the boundary pixels result in similar losses. There is a performance increase for no communication, but this advantage gets smaller the higher the workload of the layers is in one forward or backward iteration. Furthermore, the previous graphs show that the higher the workload of the layers is in one forward or backward iteration, the better the scaling. This phenomenon is reasonable because communication is needed less frequently. For a big enough workload, the hybrid MPI-OpenMP scaling is satisfying.

6.3.9 CUDA Parallelization for GPUs

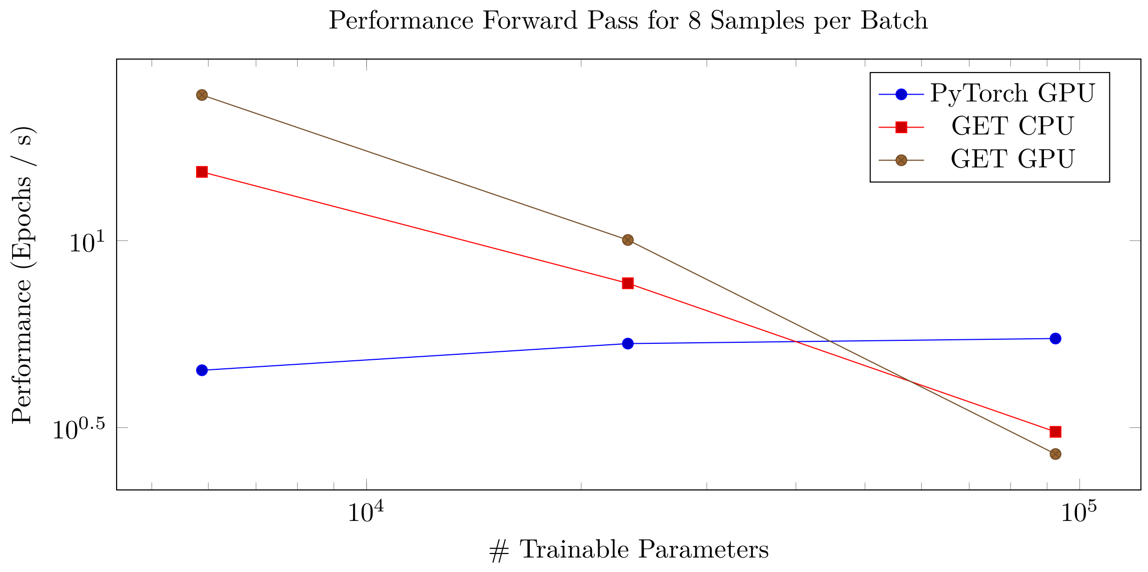
The following subsection analyzes the performance of the generated CUDA code. Only U-Nets from n = 0 to n = 2 are used since for n = 3 the code generation time is already exploding. The subsection compares the GPU and CPU performance of the GET with the PyTorch GPU performance for the complete training, the forward pass and the backward pass. First, the following graph presents the code generation times.

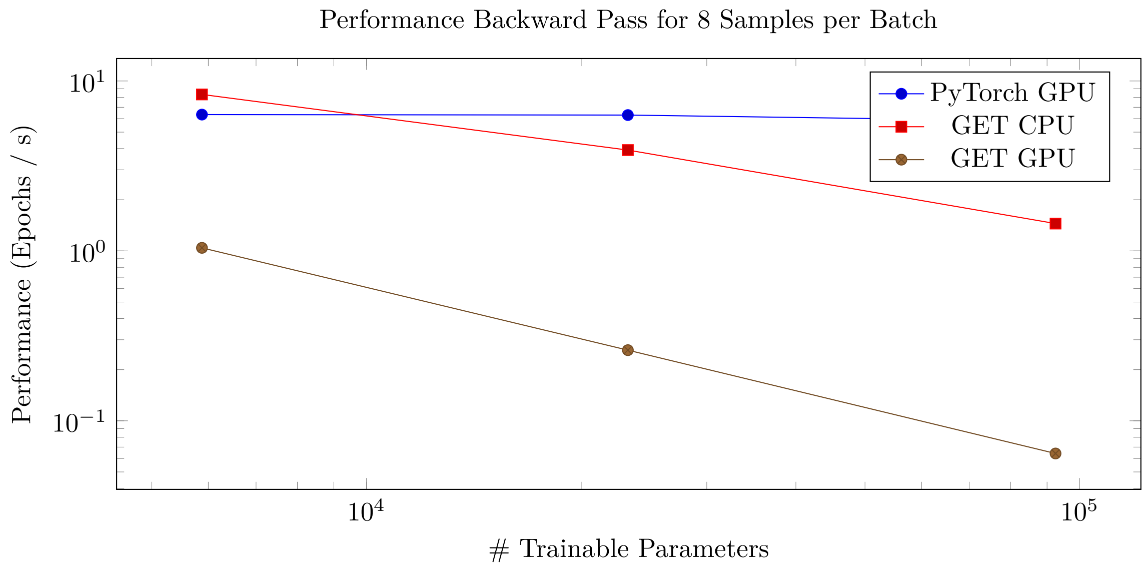


The code generation times are exploding for an increasing amount of trainable parameters, which makes it impossible to use the GET for larger U-Nets. The log scale on the y-axis has to be considered by the reader. Most of the ExaStencils code generation time is spent on the transformations "Simplify index expressions", "Handle reductions in device kernels", and "Replace assignments to reduction targets". Therefore, the reduction is a big issue of the generated code. The following graphs report the performance for eight samples per batch. 64 samples per batch already exceed the GPU memory limits, so this is omitted. The first graph compares the performance of the generated CPU and GPU codes with PyTorch for the entire training. Then, their performances are reported separately for the forward and backward pass.



It has to be taken into account that the performance unit is epochs per second, and the amount of work per epoch increases with increasing U-Net sizes. Therefore, decreasing performance for increasing U-Net sizes is plausible. For up to n = 1, the generated CPU code performs better than the PyTorch GPU code. The generated GPU code is by far worse than both the PyTorch GPU implementation and the generated CPU code. It is noticeable that the performance of the PyTorch GPU implementation stays constant for increasing U-Net sizes, which might be the case since the U-Net sizes are still moderate. The following performance graphs compare the forward and backward pass performances. These comparisons give insights into the bad performance of the generated GPU code.





Again, the log scale on the y-axis has to be considered by the reader. The forward pass of the generated code is not optimal. For small U-Nets, the generated CPU and GPU codes perform better than the PyTorch GPU implementation in the forward pass. However, this changes for n = 2. The backward pass of the generated GPU code is far worse than both PyTorch GPU and the generated CPU code. This bad performance is probably due to the inefficient implementations of the reductions since this is the main difference in computations between the forward and backward pass. The following code snippet illustrates the layer 4 code of the forward pass used for the CUDA parallelization.

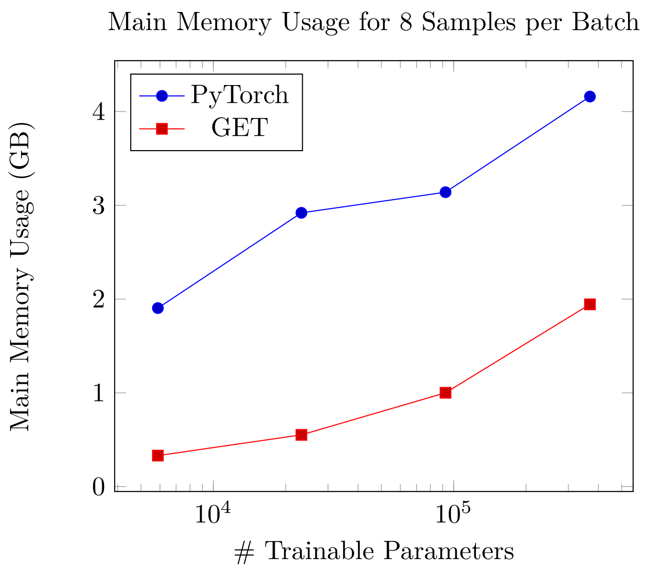
```
loop over output_image {
  Var batch : Int = 0
  repeat 8 times count batch {
    Var channel : Int = 0
    repeat 64 times count channel {
      Var channel_input : Int = 0
      output_image[batch][channel] = 0.0
      repeat 32 times count channel_input {
        output_image[batch][channel] += ( input_image@[-1, -1][batch][
            channel_input] * conv_weights[0][some_index] ) + ...
      }
      output_image[batch][channel] += conv_biases[0][120 + channel]
    }
  }
}
```

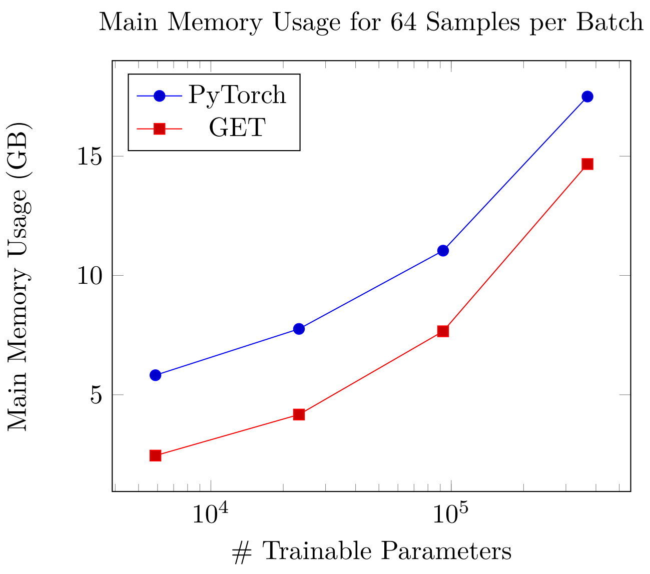
Listing 43: Convolutional layer Forward Pass.

This code snippet illustrates one problem with the CUDA code since it parallelizes only the outer loop, i.e., over the pixels of the image. $\#batches * \#channels$ output values must be computed per pixel. However, only one kernel per pixel has to perform all those computations. This approach is not the proper way of a CUDA parallelization since one kernel should operate on the smallest unit of work.

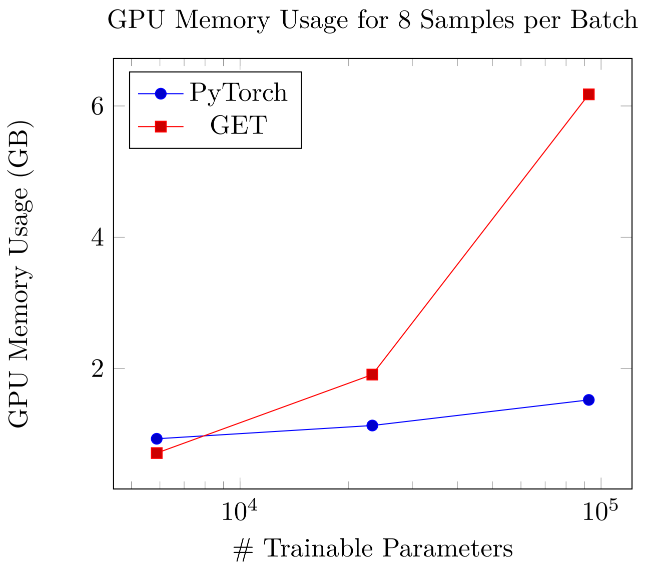
6.3.10 Main Memory Usage

As described by Ronneberger et al., the main memory consumption is often a limiting factor that forces the training of large U-Nets to use only one image per batch [23]. This limitation shows that the main memory consumption is essential. The following graphs use 32-bit floating-precision data structures, and the same optimization flags are active as described in 6.3.1. The following CPU main memory consumption is the "Maximum resident set size" reported by "/bin/time -v command" on the Ice Lake for 72 threads.





The generated codes use significantly less main memory than PyTorch on the CPU. The following graph reports the GPU memory consumption up to n = 2.



Again, the same pattern shows up as in the previous GPU analysis, which is that the generated GPU code is worse than the PyTorch implementation, also in terms of memory consumption.

7 Discussion and Conclusions

This thesis aimed to implement a code generation toolchain for generating highly optimized and parallel C++ training codes for given U-Nets. A significant training performance improvement compared with PyTorch was expected. The thesis used ExaStencils as the code generation framework. Furthermore, a Python framework called GNNESS was developed. Its task is to generate the training code for a given PyTorch model in the DSL ExaSlang. ExaStencils and GNNESS together form the GET, which can generate codes with OpenMP, hybrid MPI-OpenMP and CUDA parallelization. Furthermore, the GET supports all layers needed to generate training codes for U-Nets with similar architectures like the one introduced by Ronneberger et al. The popular "Adam" optimizer and the "Cross entropy loss" can be used. The thesis evaluated that the generated codes compute correct results using an image segmentation problem. In addition, the thesis assessed the usability and user experience. It presented how training a NN using PyTorch looks like, how it works for the GET and compares and discusses both approaches from a usability point of view. It was concluded that the user experience is comparable to PyTorch for standard training, if not easier. However, generating codes for non-standard training processes is not yet possible. The thesis presented performance comparisons on a state-of-the-art CPU node (Intel Xeon Ice Lake) and GPU (NVIDIA A100). The overhead due to the code generation times is neglectable for CPUs. However, for GPUs, the code generation time is beyond reasonable limits. The thesis evaluated the performance of the generated codes by comparing them with PyTorch. The OpenMP parallel intra-node performance results were not satisfying in the beginning due to non-consecutive memory accesses. This was avoided by changing the loop order and combining "#omp pragma" sections with "#pragma omp for ... nowait". After this optimization, the performance analysis results were promising, even if there is further optimization potential. The generated codes outperformed PyTorch for U-Nets for all investigated sizes (between 5,879 and 368,955 trainable parameters) and batch sizes by a factor of 1.84 to 25. The speedup was smaller for larger workloads per layer (more trainable parameters per layer or more samples per batch). The scaling across multiple CPU nodes using a hybrid MPI-OpenMP parallelization was promising, with a speedup of 14.9 for 64 samples per batch on 16 nodes. The GPU performance of the generated code is significantly worse than PyTorch. The main reason for this is the inefficient reduction implementation and the workload distribution.

This thesis reached the goal of implementing a code generation toolchain for generating optimized and parallel U-Net training codes. The code generation times for CPU codes remain negligibly small, even for large U-Nets, so the resulting additional effort is justifiable if it leads to higher performance. The goal of a significant performance improvement compared with PyTorch was reached for the OpenMP parallelization and the investigated U-Net sizes, which shows the potential of using code generation technologies for training U-Nets. However, the generated codes' performance is still far from the light-speed estimation, possibly leading to an even bigger speedup of the generated codes. The interpretation of these performance results must take into account that PyTorch is among the best performing deep learning frameworks. Compared to five popular deep learning frameworks (Chainer, CNTK, MXNet, PaddlePaddle, and TensorFlow), PyTorch is for three popular NN architectures (AlexNet, VGG-19, ResNet-50) either the best (in terms of throughput) or very close to the best framework [22]. As mentioned in section 2, Venkat et al. [29] used the SWIRL compiler for the generation of high-performance CPU implementations for training NNs. This approach is similar to the one taken in this thesis. They compared the performance of their generated codes with the TensorFlow-MKL implementation for different famous NN architectures like AlexNet, VGG, and GoogLeNet-v1. They achieve a similar performance with their generated codes to the TensorFlow-MKL implementation. In contrast, a significant speedup compared to the PyTorch implementation was achieved in this thesis. However, the speedup is getting smaller for larger U-Nets and more workload per layer. The thesis analyzed the performance only for small to medium-sized U-Nets (up to 368,955 trainable parameters). The speedup might be different for very large U-Nets, which limits the comparability with the results of Venkat et al. since they used much larger NNs. This thesis has mainly focused on generating CPU codes, so the GPU codes are far from optimal. The GPU performance comparison does not confirm the hypothesis that code generation improves the training compared with PyTorch. However, this hypothesis cannot be rejected either since there are obvious performance issues with the generated GPU codes. The results indicate that GPUs are far superior to CPUs in deep learning. With a theoretical peak performance of roughly 3 TFLOPs, one Intel Xeon Ice Lake CPU is way behind one NVIDIA A100 GPU with 19.5 TFLOPs. The difference gets even more extreme when using Tensor Cores that achieve 156 TFLOPs. However, as the following section explains, this is not the complete truth.

8 Outlook

Although the thesis achieved the goal of implementing a toolchain to generate U-Net training codes for given PyTorch models, there is still much work to be done. The first question that needs to be addressed is whether to continue to stick with ExaStencils as the code generation framework for U-Nets and NNs. This thesis has shown that ExaStencils is suitable for this task, and one should seriously consider using it. The second question that needs to be answered beforehand is whether work should continue on generating CPU and GPU codes or whether the focus should be exclusively on GPUs. In principle, there are good reasons to focus solely on code generation for GPUs since they are the de facto standard for training most deep learning models. However, there are also application fields where CPUs are superior:

Mittal et al. [19] provide a survey of deep learning optimizing techniques on CPUs. They report that companies like Amazon and Google use resources for optimizing deep learning applications on CPUs. Furthermore, they present the areas of strength of CPUs in the context of deep learning. According to Mittal et al., CPUs are superior to GPUs regarding the main memory capacities, which is especially relevant for 3D and 2D convolutions with large batch sizes. Furthermore, CPUs can outperform GPUs for recurrent NNs or some architectures with irregular memory accesses like InceptionNet (for more details and CPU advantages, see Mittal et al.). Chen et al. [3] approach things differently. They introduce the Sub-LInear Deep learning Engine (SLIDE), which provides multi-core parallel and optimized intelligent randomized algorithms. The idea is to sparsify most neurons selectively, i.e., using locality-sensitive hash tables to identify the neurons to be updated efficiently. Their approach outperforms a TensorFlow implementation on an NVIDIA V100 GPU using a single CPU. These application fields where CPUs are superior must be considered when deciding whether to focus purely on GPU code generation in the future.

The first problem with the CPU codes generated so far is that they store the trainable parameters in data structures on the stack. However, they should be on the heap. Otherwise, this will lead to a stack overflow for larger U-Nets. In addition, the generated initFieldsWithZeros function is not implemented in a NUMA-aware way at the moment for fields containing matrices in the cells, which should be fixed in the future. Furthermore, subsection 6.3.4 indicated that the generated code's performance on one NUMA domain is still away from being close to the light-speed estimation. Hence, there is potential for optimization, which further investigations could examine. In addition, subsection 6.3.5 showed that the generated code's scaling across multiple NUMA domains offers possibilities for improvement. The parallel efficiency dropped from roughly 74% to less than 40%, which one can partly attribute to the fact that with each reduction, the entire data structure is reduced so far and not just the slice that the current step of the backward pass changed. The scaling across multiple NUMA domains should be optimized in the future. Additionally, the performance for larger U-Nets should be examined. Furthermore, ExaStencils should natively support enclosing parallel regions (see subsection 6.3.4) if the OpenMP parallelization is not applied on the outermost of multiple nested for loops. Using the OpenMP directive "nowait" in the loop parallelization significantly reduces the parallelization overhead. Alternatively, the data structures can be changed such that the problem of non-consecutive memory accesses as described in subsection 6.3.4 is no longer an issue. Finally, there should be much more focus on the possibility of cross-layer optimization, which can significantly reduce the runtime through less data transfer [29]. The thesis only marginally considered these optimizations.

If the focus is on generating GPU codes, the question should be answered to what extent code generation technologies can improve GPU performance significantly compared to frameworks like PyTorch. The thesis did not show this since the generated code's performance was worse than that of PyTorch. As soon as one answers this question in the affirmative, the focus should be on analyzing and optimizing the generated CUDA code. Changing the reduction implementation and the workload per kernel call will likely offer significant optimization potential.

References

[1] Daniel Arndt, Wolfgang Bangerth, Bruno Blais, Marc Fehling, Rene Gassmöller, Timo Heister, Luca Heltai, Uwe Köcher, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Pelteret, Sebastian Proell, Konrad Simon, Bruno Turcksin, David Wells, and Jiaqi Zhang. The `deal.II` library, version 9.3. *Journal of Numerical Mathematics*, 29(3):171–186, 2021. URL: https://dealii.org/deal93-preprint.pdf, doi:10.1515/jnma-2021-0081.

[2] Katharina Breininger. Convolutional neural networks, 2020. Deep learning exercise notes. URL: https://univis.fau.de/form?__s=2&dsc=anew/module_view&print=1&anonymous=1&dir=tech/IMMD/IMMD5&mod=tech/IMMD/IMMD5/23&ref=lecture&sem=2020w&__e=50.

[3] Beidi Chen, Tharun Medini, James Farwell, Charlie Tai, Anshumali Shrivastava, et al. Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *Proceedings of Machine Learning and Systems*, 2:291–306, 2020.

[4] François Chollet et al. Keras. https://keras.io, 2015.

[5] Torch Contributors. Pytorch documentation. Accessed on 15.01.2022. URL: https://pytorch.org/.

[6] Torch Contributors. Crossentropyloss, 2019. Accessed on 02.01.2022. URL: https://pytorch.org/docs/1.9.1/generated/torch.nn.CrossEntropyLoss.html.

[7] Torch Contributors. Sgd, 2019. Accessed on 20.12.2021. URL: https://pytorch.org/docs/stable/generated/torch.optim.SGD.html#torch.optim.SGD.

[8] NVIDIA Corporation. Nvidia a100 tensor core gpu data sheet, 2021. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf.

[9] Sara Faghih-Naini, Sebastian Kuckuk, Vadym Aizinger, Daniel Zint, Roberto Grosso, and Harald Köstler. Quadrature-free discontinuous galerkin method with code generation features for shallow water equations on automatically generated block-structured meshes. *Advances in Water Resources*, 138:103552, 2020.

[10] Georg Hager. Write-allocate evasion has finally arrived at intel – or has it?, 2021. Accessed on 24.12.2021. URL: https://blogs.fau.de/hager/archives/8997.

[11] Georg Hager, Gerhard Wellein, and Jan Eitzinger. Simple performance modeling: The roofline model, 2021. Node-level performance engineering lecture notes. URL: https://moodle.rrze.uni-erlangen.de/course/view.php?id=274&username=guest&password=guest.

[12] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] Sebastian Kuckuk. *Automatic code generation for massively parallel applications in computational fluid dynamics*. FAU University Press, 2019.

[14] Christian Lengauer, Sven Apel, Matthias Bolten, Shigeru Chiba, Ulrich Rüde, Jürgen Teich, Armin Größlinger, Frank Hannig, Harald Köstler, Lisa Claus, et al. Exastencils: Advanced multigrid solver generation. In *Software for Exascale Computing-SPPEXA 2016-2019*, pages 405–452. Springer, Cham, 2020.

[15] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. Automatic code generation of convolutional neural networks in fpga implementation. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 61–68. IEEE, 2016.

[16] Andreas Maier. Activation functions and convolutional neural networks, 2020. Deep learning lecture notes. URL: https://univis.fau.de/form?__s=2&dsc=anew/module_view&print=1&anonymous=1&dir=tech/IMMD/IMMD5&mod=tech/IMMD/IMMD5/23&ref=lecture&sem=2020w&__e=50.

[17] Andreas Maier. Feed forward neural networks, 2020. Deep learning lecture notes. URL: https://univis.fau.de/form?__s=2&dsc=anew/module_view&print=1&anonymous=1&dir=tech/IMMD/IMMD5&mod=tech/IMMD/IMMD5/23&ref=lecture&sem=2020w&__e=50.

[18] Andreas Maier. Segmentation and object detection - part 1, 2020. Accessed on 01.01.2022. URL: https://towardsdatascience.com/segmentation-and-object-detection-part-1-b8ef6f101547.

[19] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. A survey of deep learning on cpus: opportunities and co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[20] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G Spampinato, and Markus Püschel. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85. IEEE, 2014.

[21] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. Cats and dogs. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3498–3505. IEEE, 2012.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[23] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[24] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[25] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. Exaslang: A domain-specific language for highly scalable multigrid solvers. In *2014 Fourth international workshop on domain-specific languages and high-level frameworks for high performance computing*, pages 42–51. IEEE, 2014.

[26] Christian Schmitt Sebastian Kuckuk and Stefan Kronawitter. Exastencils repository. URL: https://i10git.cs.fau.de/exastencils/release.

[27] Emma Söderberg, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming*, 78(10):1809–1827, 2013.

[28] The ImageMagick Development Team. Imagemagick. URL: https://imagemagick.org.

[29] Anand Venkat, Tharindu Rusira, Raj Barik, Mary Hall, and Leonard Truong. Swirl: High-performance many-core cpu code generation for deep neural networks. *The International Journal of High Performance Computing Applications*, 33(6):1275–1289, 2019.

[30] Julien Vitay, Helge Ülo Dinkelbach, and Fred H Hamker. Annarchy: a code generation approach to neural simulations on parallel hardware. *Frontiers in neuroinformatics*, 9:19, 2015.

[31] Esin Yavuz, James Turner, and Thomas Nowotny. Genn: a code generation framework for accelerated brain simulations. *Scientific reports*, 6(1):1–14, 2016.

[32] Lu Yuan, Dongdong Chen, Yi-Ling Chen, Noel Codella, Xiyang Dai, Jianfeng Gao, Houdong Hu, Xuedong Huang, Boxin Li, Chunyuan Li, et al. Florence: A new foundation model for computer vision. *arXiv preprint arXiv:2111.11432*, 2021.

[33] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

Acronyms

Adam	Adaptive Moment Estimation.
AST	Abstract Syntax Tree.
ASTs	Abstract Syntax Trees.
CNN	Convolutional Neural Network.
CNNs	Convolutional Neural Networks.
CPU	Central Processing Unit.
CPUs	Central Processing Units.
DSL	Domain-Specific Language.
FAU	Friedrich-Alexander-Universität Erlangen-Nürnberg.
GET	GNNESS + ExaStencils Toolchain.
GHODDESS	Generation of Higher-Order Discretizations Deployed as ExaSlang Specifications.
GNNESS	Generation of Neural Networks deployed as ExaSlang Specifications.
GPU	Graphics Processing Unit.
GPUs	Graphics Processing Units.
ILSVRC	ImageNet Challenge.
NN	Neural Network.
NNs	Neural Networks.
ReLU	Rectified Linear Unit.
SGD	Stochastic Gradient Descent.
w.r.t	with respect to.

Listings

1 Pseudocode for the convolution forward pass. 16
2 Pseudocode for the convolution backward pass. 17
3 Pseudocode for the transposed convolution forward pass. 19
4 Pseudocode for the transposed convolution backward pass (gradient w.r.t. the lower layer). 20
5 Pseudocode for the transposed convolution backward pass (gradient w.r.t. the weights). 21
6 Minimal layer 4 code example. 23
7 Knowledge file example. 24
8 Platform file example. 24
9 Definition of the function m. 25
10 First part of the globalVariableStorage.py code snippet. 29
11 Second part of the globalVariableStorage.py code snippet. 29
12 Third part of the globalVariableStorage.py code snippet. 29
13 Fourth part of the globalVariableStorage.py code snippet. 30
14 Fifth part of the globalVariableStorage.py code snippet. 30
15 First part of the fieldStorage.py code snippet. 31
16 Second part of the fieldStorage.py code snippet. 31
17 Third part of the fieldStorage.py code snippet. 31
18 Fourth part of the fieldStorage.py code snippet. 32
19 Fifth part of the fieldStorage.py code snippet. 32
20 Code snippet from node.py. 33
21 Code snippet from function.py. 33
22 First part of the functionCollection.py code snippet. 33
23 Second part of the functionCollection.py code snippet. 34
24 First part of the expressions.py code snippet. 34
25 Second part of the expressions.py code snippet. 34
26 First part of the statements.py code snippet. 34
27 Second part of the statements.py code snippet. 35
28 Code snippet from application.py. 35
29 Layer 4 code of the "Application" function. 36
30 ReLU code snippet from forward.py. 36
31 Layer 4 code of the ReLU forward pass. 37
32 First part of the layer4.py code snippet. 37
33 Second part of the layer4.py code snippet. 38
34 Third part of the layer4.py code snippet. 38
35 Fourth part of the layer4.py code snippet. 38
36 Minimal example of how to train a NN using PyTorch. 41
37 Staring the PyTorch training. 42
38 Configuration file for the GET. 42
39 Staring the GET. 43
40 Convolutional layer forward pass of the first code generation approach. 53
41 Convolutional layer forward pass of the second code generation approach. 53
42 Convolutional layer forward pass to be analysed. 60
43 Convolutional layer Forward Pass. 65

List of Figures

1 Image segmentation task. Raw image of cells on the left, segmentation mask on the right. Pixels that belong to the first segment (cells) are white, those who belong to the second segment (background) are black [23]. 11
2 U-Net architecture introduced by Ronneberger et al. Blue boxes correspond to multi-channel feature maps. Grey numbers on top of the boxes represent the number of channels. The feature map sizes are denoted on the left of the boxes. Arrows indicate the data flow between different states, beginning at the input image tile and ending at the output segmentation map [23]. 12
3 Backpropagation algorithm for one NN layer. x_1 is the input of the layer, x_2 denotes its trainable parameters, $\hat{y}$ represents the output. The blue circle stands for one complete layer. L is the loss of the NN [16]. 14
4 Forward pass of a convolutional layer. Input on the left, kernels in the center, output on the right. The meaning of the red lines is explained below (modification of [16]). 15
5 First part of the backward pass of a convolutional layer. Gradient w.r.t. the output of the previous layer. New error tensor on the left, kernels in the center, old error tensor on the right. The meaning of the red lines is explained below (modification of [2]). 16
6 Second part of the backward pass of a convolutional layer. Gradient w.r.t. the trainable parameters. Input image on the left (with one padding row on each side in x and y direction), error tensor in the center, gradient w.r.t. the trainable parameters on the right. The meaning of the green lines is explained below (modification of [2]). 17
7 Forward pass of an average and max pooling layer. 18
8 Backward pass of an average and max pooling layer. 19
9 Forward pass of a transposed convolutional layer (kernel size: 2x2, stride: 2x2). . . . 19
10 Backward pass of a transposed convolutional layer (kernel size: 2x2, stride: 2x2). Error tensor denotes the new error tensor. 20
11 ExaSlang layers of abstraction [14]. 22
12 AST of the previously defined method m() [27]. 25
13 Workflow of the code generation toolchain used in this thesis, starting with a PyTorch model and config file (left) and ending with a trained NN (right). 27
14 Cat image (left-hand side) and its pixel-level segmentation into three segments (right-hand side) from the Oxford-IIIT pet data set [21]. 40
15 U-Net architecture for the performance evaluations. Blue boxes correspond to multi-channel feature maps. Grey numbers on top of the boxes represent the number of channels, which depends on an integer n. The feature map sizes are denoted on the left or below the boxes. Modification of [23]. 44

List of Tables

1 Comparison between multigrid algorithms and U-Nets. Each line contains a similar operation in the terminology of multigrid (left) and U-Nets (right). 26
2 Characteristics of one Intel Xeon Platinum 8360Y node. 45
3 Characteristics of one Intel Xeon E5-2630 v4 node. 46
4 Characteristics of one NVIDIA A100. 46
5 Software configuration of the Ice Lake node. 46
6 Software configuration of the Broadwell node. 46
7 Software configuration of the A100 GPU node. 47
8 Runtime comparison of the U-Net layers. 53
9 Characteristics of one Ice Lake SP core [11]. 60
10 Performance measurements for the first convolution forward pass. 61
11 Performance measurements for the second convolution forward pass. 62