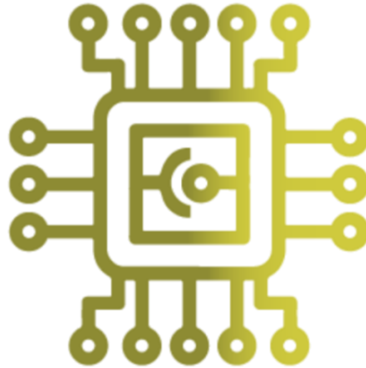

Seamless design of smart edge processors

GRANT AGREEMENT NUMBER: 101070374

Deliverable D5.1

Constraints and opportunities definition

Disclaimer
This project has received funding from the European Union's Horizon 2021 research and innovation programme under grant agreement No 101070374. This document has been prepared for the European Commission, however, it reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

CONVOLVE

Title of the deliverable	Constraints and opportunities definition
WP contributing to the deliverable	WP 5
Task contributing to the deliverable	Task 5.1
Dissemination level	PU – Public
Due submission date	30/04/2023
Actual submission date	03/05/2023
Author(s)	Alexandra Jimborean Kunwar Grover Ravikiran R Victor Jung Josse Van Delm Gianna Paulin Sasha Lopoukhine Sven Argo Sam Ainsworth Tobias Grosser Jose Luis Abellan
Internal reviewers	Jan Richter-Brockmann Taouil Mottaqiallah

Document Version	Date	Change
V0.1	24/02/2022	Table of content and main document structure
V0.2	14/03/2023	Table of content aligned to the template and assignments
V0.3	06/04/2023	Style formatting
V0.4	09/04/2023	New template

Table of Contents

Deliverable Summary 6

1. Introduction 6

2. High-level Description of the Objectives 6

2.1. WP Objectives 6

2.2. WP Contribution to CONVOLVE'S Objectives 7

3. State of the Art in the Neural Networks Compilation Flows 8

3.1. Frameworks 8

3.1.1. MLIR 8

3.1.2. PyTorch 9

3.1.3. Torch-MLIR 9

3.1.4. TorchScript 9

3.1.5. LinAlg (on Tensors and Buffers) 9

3.1.6. Loops(Affine/SCF) 10

3.1.7. LLVM 10

3.1.8. RISC-V 10

3.2. Neural Networks Deployment 11

3.3. Neural Networks Optimizations 11

3.3.1. Quantization 12

3.3.2. Layout Optimizations 12

3.3.3. Pruning 13

3.3.4. Multithreading 13

3.3.5. Transformation 13

3.4. Kernel Optimizations 13

3.4.1. Tiling 13

3.4.2. Operator Fusion 14

3.5. Optimization Frameworks 14

3.5.1. Halide for High-Performance Image and Signal Processing 15

3.5.2. Tensor Virtual Machine (TVM) 15

3.5.3. Tensor Comprehensions 16

3.5.4. Dory 16

3.5.5. OpenXLA : StableHLO, XLO, IREE 17

3.6. Security in Compilation 17

3.6.1. Side-Channel Attacks and State-of-the-Art Countermeasures 17

3.6.2. Reverse Engineering 19

3.6.3. Secure Compilation in CONVOLVE 20
4. High-level Description of the Target Architecture and its Accelerators 20
4.1. The PULP platform 21
4.2. Programming Interface 22
4.2.1. Accelerators 23
5. Roadblocks 24
6. Reseach 27
6.1. Compilation Pipeline 27
6.2. Research on Interfacing with the Accelerators 29
6.3. Dependencies with other WPs 30
6.4. Use Cases Requirements Addressed by the WP 31
6.4.1. Use-Case 1: Speech Quality and Denoise 32
6.4.1.1. Description 32
6.4.1.2. Corresponding Security Concerns 33
6.4.1.3. Reference to the Complied Code 34
6.4.1.4. Characterization of the Use Case 34
6.4.2. Use-Case 2: Audio Detection and Tracking 35
6.4.2.1. Description 35
6.4.2.2. Conclusion 36
6.4.3. Use-Case 3: Image Processing 36
6.4.3.1. Description 36
6.4.3.2. Characterization of the Use Case 36
6.4.4. Use-case 4: Satellite Image Segment 38
6.4.4.1. Description 38
6.4.4.2. Conclusions 38
6.5. Description of the Contributions to the Demos 38
7. Plans 39
8. Reference 40

Deliverable Summary

This document presents first a high-level description of the objectives of the work-package, focusing on the compilation-flow and mapping of the neural network applications to the accelerators developed in this project. We present then the state-of-the-art tools employed in compiling and optimizing neural networks and show their limitations. We also review security concerns and constraints, together with some compile-time methods for mitigating them.

Next, we turn the attention to the target hardware, we describe the target hardware platforms and the envisioned accelerators to be developed in CONVOLVE, together with their characteristics. The hardware characteristics will be leveraged by the compiler when mapping applications to accelerators.

Generating optimized code and mapping the applications to the target accelerators is a challenging task. We describe the compilation-flow, the tools, and the open-source infrastructure (such as LLVM, MLIR) that we plan to employ and present several scenarios for interfacing with the accelerators. To ease this process, we aim to combine automatic code generation and optimization with expert intervention, through a "grey-box" compilation flow, that opens-up the compiler decisions to the expert. We conclude with the research roadmap envisioned in the work-package.

1. Introduction

This document "Constraints and opportunities definition" is a deliverable of the Work package No. 5 "Transparent and compositional programming flow", task T5.1 "Analysis of application and hardware constraints and opportunities and Software Design" under the task lead of UMU, sets out the deliverable D5.1 "Constraints and opportunities definition".

2. High-level Description of the Objectives

We first provide an overall context by presenting the objectives of this work package and how these objectives contribute to the CONVOLVE objectives.

2.1. WP Objectives

The objectives of WP5 are as follows:

- Empower domain and hardware experts to build tailored deep-learning compiler

  Reaching peak-performance requires the experience of application and hardware experts as input for optimizing the code. Hence, we will reduce the barrier between experts and compilers through a novel grey-box compiler experience where domain and hardware experts are placed in control of the lowering of their programs. As part of this grey-box compiler experience, we will analyse pre-existing deep learning compiler stacks and make it easier for domain-experts to interact with and modify these stacks. The

stacks and locations at which we 'grey-box' them will be strongly influenced by the input of the CONVOLVE experts.

- Improve our understanding of the CONVOLVE applications and hardware, also in tandem
  We will offer tooling and analysis of the CONVOLVE applications (e.g., automatic static and dynamic analysis) to get a good understanding of both the CONVOLVE applications as well as the effectiveness of their hardware mapping. To gain this understanding, we will develop tailored analyses for the CONVOLVE applications and will offer the ability to connect hardware analysis tools to analyse the execution of applications on CONVOLVE hardware. We will also contribute towards the actual analysis of the CONVOLVE applications.

- Support domain to hardware mapping with automation

  The compiler technology has the ability to automate the optimization of programs. We will analyse the set of optimizations that domain-experts apply manually and offer technology and tooling to support this process through automation. One of these areas of automation is the exploration of the design space of potential accelerator mapping strategies. One key technology in this space will be peephole optimizations, which can be defined by experts and which we will then automatically apply and search. Additionally, we will investigate optimizations that can be performed automatically by the compiler but would be too cumbersome to apply manually.

- Optimally use pre-existing software stacks (e.g., MLIR, Dory, ...)

  Deep learning compiler stacks are numerous, many are particularly strong in certain areas, and often an entire set of tools is needed when mapping from an application to hardware. We aim to embrace this ecosystem, reuse code as much as possible, and also identify and minimize the impedance mismatches between existing systems, e.g., established deep learning stacks are well-optimized for DNN to CPU/GPU compilation yet lack support for many custom accelerators. Yet, ETH and others have excellent software stacks tailored to their hardware, but they are not connected to latest software stacks (e.g., MLIR). We will use our expertise to reduce this gap and offer compilation support for CONVOLVE.

2.2. WP Contribution to CONVOLVE'S Objectives

The compiler work package WP5 provides the glue that connects the CONVOLVE neural network domain and hardware experts and it is critically important to enable the CONVOLVE objectives.

- 10x faster code

  By offering a grey-box compiler experience, where domain and hardware experts take an active part in optimizing their applications, our compiler enables the use of optimizations tailored for the specific application and hardware combinations relevant to CONVOLVE.

Such optimizations are not limited by generic compiler optimization passes but can be as good as hand-optimized code. In addition, our support for analysing the neural networks and their (predicted) performance on the CONVOLVE hardware will empower our experts to take better accelerator mapping decisions. As a result, we can choose between the best of two worlds – automation through compile-time optimizations and mapping and peak-performance offered by the expert-knowledge available from our CONVOLVE partners.

- Optimizations and compilation flow to fully leverage the accelerators

  We fully leverage the accelerators and hardware in CONVOLVE by embracing existing technology, evolving and rethinking it in the light of latest research results around MLIR, and considering the pre-existing technology stacks in the CONVOLVE team. We also offer optimizations, by increasing the level of automation in the hardware mapping workflow.

- Infrastructure for Machine Learning engineers to estimate efficiency of NNs

  Our work on understanding of the CONVOLVE applications in the context of the CONVOLVE hardware provides static analysis that offer insights that are hard to reason about manually. Thanks to the Grey-box compiler we develop, these insights will be presented in the exact context of the NNs as they are mapped to the accelerators, such that machine learning engineers can take action based on this feedback.

3. State of the Art in the Neural Networks Compilation Flows

We first give an overview of existing frameworks for compiling deep neural networks and subsequently discuss the roadblocks for using them in CONVOLVE.

3.1. Frameworks

Before we dive deeper into the design of our compiler stack, let us briefly introduce relevant background. We introduce the common tools employed in compiling NNs.

3.1.1. MLIR

MLIR (https://mlir.llvm.org/) is a framework to build compiler IRs (Intermediate Representation). MLIR allows one to define their own set of operations, own type system and benefit from pass management, diagnostics, multi-threading, serialization/deserialization, etc. MLIR does this in form of a dialect, where a dialect is a group of operations defined by the user together with their type system. Dialects allow the user to write optimisations at the most relevant level of abstraction. They can also be interleaved with each other.
By using MLIR, we can decouple the teams working on different parts of the compiler. The experts in neural network optimisation can write the optimisations at a high level of abstraction, and the experts in accelerator utilisation can write optimisations at a lower level.

Using a single framework for the different levels of abstraction lets us write analyses that work throughout the compilation phases. For example, for the estimates of energy use of a given neural network, we can compile it down using the various dialects in MLIR. We can then propagate the energy use of single instructions in the output backwards, annotating the higher-level instructions with aggregate energy use. Similarly for cycle counts, etc.

3.1.2. PyTorch

PyTorch (pytorch.org), a widely used framework to express deep neural networks in Python, will be used as input to our compilation flow. Several of the use cases our application partners provided are available as PyTorch implementations. The dynamic and extensible architecture of PyTorch that made it popular for machine learning research is particularly suitable for CONVOLVE use-cases.

In addition to the PyTorch input file, we will also require an example input to derive tensor shape information (e.g., run shape inference), which is necessary to fully specify the pipeline.

3.1.3. Torch-MLIR

The Torch-MLIR (https://github.com/llvm/torch-mlir) project acts as a bridge between PyTorch and MLIR. It aims to provide first class compiler support from the PyTorch ecosystem to the MLIR ecosystem. It provides an MLIR dialect, lowerings to operations built into the MLIR framework. By leveraging the work already done by that open-source community, we can get started quickly optimising and analysing the neural networks that have been proposed for the CONVOLVE use-cases.

3.1.4. TorchScript

TorchScript is a machine readable and structured serialization format for PyTorch and serves as connection layer between PyTorch and the MLIR dialects. TorchScript and the conversion from PyTorch to TorchScript is maintained by the PyTorch project. Hence, we can rely on it as well-documented, complete, and stable component of our infrastructure. The conversion between Torch and TorchScript also does not involve complex semantic changes, which means we do not risk to lose information by relying on TorchScript. Finally, TorchScript is recognized as an interfacing abstraction between PyTorch and MLIR. In particular, the Torch-MLIR project imports TorchScript and translates it into an MLIR TorchScript dialect, at which level we can convert TorchScript into any MLIR dialect using standard MLIR transformations. Torch-MLIR also performs static analyses such as shape inference, data type inference and value semantics maximization is performed, that are necessary for effective code generation further down the pipeline. Torch-MLIR offers itself a translation from TorchScript into MLIR's linear algebra abstraction (LinAlg), which we use as a next step in our pipeline.

3.1.5. LinAlg (on Tensors and Buffers)

We use MLIR's linear algebra dialect LinAlg (https://mlir.llvm.org/docs/Dialects/Linalg/) as the high-level representation of tensor computations. LinAlg is a core MLIR dialect that implements a structured abstraction of linear algebra operations on both tensors and buffers. Tensors in LinAlg have value semantics, which means they are newly created by operations and afterwards remain read-only. Buffers on the other side are multi-dimensional memory objects which can be modified in place. LinAlg defines various linear algebra operations over tensors (or the corresponding buffers), yet it also offers a generalization of such operations with its `linalg.generic`. The `linalg.generic` operation[1] can express a family of custom operations, where the elementwise computation is arbitrary user-defined scalar arithmetic and the overall structure of the operation matches the general structure of a linear algebra operation.

When translating LinAlg to hardware, two variants of the LinAlg dialect are used. First, LinAlg over tensors is used as a convenient interface to higher-level abstractions which typically are closer to mathematical notation where in-place modifications are typically avoided for clarity. Second, LinAlg is translated to operate on buffers (so-called memrefs), by running a bufferization pass. The bufferization pass will decide on memory allocation and reuse and brings LinAlg closer to the explicit memory management that code running on actual hardware typically requires. Finally, LinAlg lowers to structured control flow consisting explicit loops, arithmetic operations, as well as loads and stores to memory.

3.1.6. Loops(Affine/SCF)

This level represents explicit memory access and loops as a dialect. Affine dialect restricts loops and the memory access inside to be affine in nature. The SCF (Structured Control Flow) dialect allows for more general loops. Affine is a subset of SCF and can be lowered to SCF.

These dialects further lower to LLVM.

3.1.7. LLVM

LLVM (llvm.org) is an open-source compiler framework widely used in the industry. It has a big community, adding optimisations at many stages of its own pipeline. We aim to leverage both MLIR and LLVM for optimisations specific to the CONVOLVE project, in addition to benefiting from the existing optimisations "for free".

The MLIR project is a part of the LLVM project, and leverages LLVM as the final stage of the compilation. LLVM code generation is available for many architectures, including RISC-V.

3.1.8. RISC-V

RISC-V (riscv.org) is an open standard Instruction Set Architecture (ISA), that is designed to be extensible. The ISA makes it easy to add instructions to leverage novel accelerators and has a small and relatively simple core set of instructions, making it an attractive target for compilation. We use the LLVM code generation framework to generate RISC-V code.

3.2. Neural Networks Deployment

Many different approaches exist to deploy various types of neural networks across various flows. Typically, two flows are supported for neural network programs:
Training (+ inference) => in = NN graph, Dataset , Out => Weights, Prediction
Inference => in = NN graph, Input data, Weights, Out => Prediction
Typically, hardware acceleration of kernels in these approaches is achieved by:

- Handwriting the entire application (not scalable)
- Using hardware-specific hand-optimized kernel libraries called by a runtime system (E.g. using CuDNN for NVIDIA GPUs in PyTorch, or CMSIS-NN on ARM-uControllers)
- Using hardware-specific parametrizable kernel libraries that can be specialized by template expansion and which are run on a runtime (E.g. CUTLASS for NVIDIA, or Dory for GAP8)
- Full-blown automatic kernel library generation (E.g. TVM Autotuning, TensorComprehensions)

Runtimes typically support two modes of operation:

- Static graph execution: The NN graph is parsed ahead-of-time and the entire network execution is planned (cfr. Classic Tensorflow). This allows for thorough compiler optimization, but does not allow for the execution of dynamic neural networks.
- Dynamic graph execution: The NN graph is parsed at runtime (which can entail a high runtime overhead) and optimized kernels are called at runtime (cfr.PyTorch, Tensorflow Eager mode). Some parts of the kernels can also be deferred to the runtime phase (e.g., dynamic shapes). This complicates optimization.

In an edge/embedded-deployment one typically must account for:

- *Low memory:* DRAM accesses can be energy-hungry. To increase the energy efficiency, one must place as much of the program as possible, including interfacing and weights storage, onto on-chip SRAM. This also entails that one needs to quantize one's workloads, as lower precision data requires less storage.
- *None or very limited (RT)OS support*. Typically, there is no support for virtual memory, multithreaded execution, nor all C standard library components. Performing RPC (Remote Procedure Calls)(necessary for autotuning) on such a device can be challenging.
- *No hardware caches.* Memories are implemented as scratchpads which require explicit memory movement by the programmer/compiler.
- *Limited support for C libraries.* Typical microcontrollers interface with various hardware components through a myriad of libraries, typically implemented in C. For this, some tools, such as (micro)TVM for example, can emit C code.

3.3. Neural Networks Optimizations

Compile-time optimizations for neural networks involve techniques that are applied during the compilation of neural network models, with the goal of improving their efficiency and performance. Below are a few examples.

3.3.1. Quantization

This technique involves representing the weights and activations of a neural network with a reduced number of bits (e.g., 8 bits instead of 32 bits) to reduce memory usage and increase computational efficiency. This can be done during the compilation process.

There are several challenges associated with quantization for edge devices. One of the main challenges is to find the right balance between reducing the precision of the weights and activations and maintaining the accuracy of the network. In some cases, reducing the precision too much can lead to a significant drop in accuracy, particularly for complex networks.

Another challenge is to ensure that the quantization process does not introduce too much noise or distortion into the network. This can be particularly important for applications such as computer vision or speech recognition, where small changes in the input data can have a significant impact on the output.

To determine the optimal level of quantization, it is common to perform a sensitivity analysis on a representative subset of the training data, measuring the network's accuracy for different quantization levels. This can help identify the optimal quantization level that achieves the desired accuracy while minimizing memory and computational requirements.

For Quantization we can make use of the QuantLab framework, which allows for arbitrary quantization support and exports to Dory, which is compatible with custom ONNX files. It also supports Post Training Quantization (PTQ) and Quantization Aware Training (QAT). Additionally, the framework supports several quantization schemes, including PACT, Bayesian Bits, PROFIT, STE, and Additive Noise Annealing (ANA) [1].

3.3.2. Layout Optimizations

The layout of the data used by neural networks can affect their performance. By optimizing the layout of data in memory, neural network computations can be made more efficient. For example, data can be arranged in memory to ensure that it is stored in contiguous blocks, which can reduce memory access time.

The reason for this is that modern processors have a hierarchy of memory caches that store recently accessed data to reduce the time needed to fetch data from main memory. When accessing data that is stored contiguously in memory, the cache can prefetch data and reduce the likelihood of cache misses. Cache misses can be expensive as they require the processor to access slower memory, such as main memory or disk storage.
Therefore, it is crucial to optimize the data storage layout in DNNs to minimize cache access penalties. This can be achieved by using memory allocation techniques that promote data locality, such as allocating memory in a contiguous block, reordering the dimensions of the data to maximize cache reuse, and using specialized data structures optimized for cache performance.

3.3.3. Pruning

Pruning involves removing unnecessary connections and weights in a neural network to reduce its size and improve its efficiency. Pruning can be done during the compilation process by identifying and removing redundant connections and weights that don't contribute significantly to the network's performance.

There are several methods for determining which connections in a neural network should be pruned. One method is weight magnitude-based pruning, where small weights are removed based on their magnitude, assuming they have a negligible impact on performance. Another approach is activation-based pruning, where neurons are ranked by importance and those with the smallest impact on the network's output are removed. Structured pruning involves removing entire neurons or groups of neurons. Reinforcement learning-based pruning trains an agent to identify connections that can be pruned while minimizing the impact on performance. A combination of these methods may be used, with trial-and-error needed to determine the optimal strategy, since effectiveness can depend on the network architecture and dataset used.

3.3.4. Multithreading

Multithreading involves dividing the computation of a neural network across multiple processors or devices. By distributing the computation across multiple devices, the overall computation time can be reduced. This can be done during the compilation process by identifying opportunities for parallelism in the neural network computation.

3.3.5. Transformation

DNN calculation can sometimes be further sped up by applying computational transformations to the data to reduce the number of (typically expensive) multiplications, while still delivering the same bit-wise result. The objective is to improve performance or reduce energy consumption although this can come at the cost of more intermediate results, an increased number of additions, and a more irregular data access pattern. One such example is the Winograd's algorithm.

3.4. Kernel Optimizations

Data in a given input space is transformed to another space using kernel functions. One can think of neural network layers as non-linear maps doing these transformations, i.e. kernels. Kernel optimization is a technique used in neural networks to improve their performance and efficiency by optimizing the computation of layers, which are one of the key building blocks of neural networks. There are different types of kernel optimization techniques. Below are a few examples.

3.4.1. Tiling

Convolutional neural networks (CNNs) commonly involve an inner-product computation that multiplies a filter matrix with an input feature map. To optimize this computation, it is often ordered to ensure reuse of elements of the filter matrix, which can be limited by the size of

available memory. Tiling and blocking are techniques developed to address this issue by dividing the input feature map and filter matrix into smaller tiles or blocks that fit in memory and allow for better element reuse. Parallelizing tiling involves distributing tiles across multiple compute units, such as CPU cores or GPUs, to improve overall performance.

Tiling can also be combined with other optimization techniques, such as loop unrolling and memory blocking, and can be automated through the use of polyhedral models and SAT solvers in compiler frameworks like LLVM and GCC. The polyhedral model maps computations to a multi-dimensional space to analyse and optimize loop nests and memory access patterns. The SAT solver generates an optimal tiling and scheduling strategy to minimize memory access overheads and maximize computational efficiency. This approach has shown significant improvements in performance and energy efficiency in various applications, including image processing, machine learning, and scientific simulations.

3.4.2. Operator Fusion

Operator fusion is a technique that groups together multiple operations in a deep learning model into one fused operator-group, represented as a directed acyclic graph (DAG). This grouping reduces the amount of memory needed to store intermediate feature maps, which can be a significant bottleneck in deep learning accelerators. State-of-the-art deep neural networks (DNNs) have many parameters and large feature maps that cannot fit in the memory of small devices such as IoT and edge devices. Hence, these DNNs need to access memory located outside the device (called off-chip memory), which is slower and energy expensive. In fact, accessing off-chip memory is often the slowest and most energy-consuming part of running DNNs on edge devices, and can limit the speed at which they can operate.

Within a fused operator-group, intermediate feature maps can be consumed immediately after being produced by scheduling in advance the corresponding operations of the consuming operators. This reduces the need to evict the intermediate feature maps to off-chip memory and re-load them from main memory before starting the next operation. This process can be repeated for each fused operator-group, thereby reducing the amount of data movement between on-chip and off-chip memory. By reducing the amount of memory accesses needed for intermediate feature maps, the operator fusion technique can significantly improve the performance of deep learning accelerators and reduce the memory access overhead. This technique is especially useful for existing deep learning accelerators that have limited memory resources and can result in significant improvements in energy efficiency, execution time, and performance.

3.5. Optimization Frameworks

Optimization frameworks in neural networks are tools that enable efficient and effective implementation of neural networks. They focus on optimizing the performance of neural network models, making them faster and more accurate, while minimizing resource consumption. Below are brief explanations of some optimization frameworks with examples:

3.5.1. Halide for High-Performance Image and Signal Processing

Halide is a domain-specific language and compiler that is designed to make it easier for developers to write high-performance image and signal processing applications. One of the key features of Halide is its separation of the algorithm specification from the scheduling and optimization of the computation.

In Halide, the user writes the algorithm as a sequence of pure functions that operate on multi-dimensional arrays. These pure functions describe the computation in a way that is independent of any particular schedule or tiling strategy. The user then provides separate annotations that describe how the computation should be scheduled and optimized, such as loop unrolling, loop tiling, and vectorization.

The Halide compiler then uses this information to automatically generate efficient code that is optimized for the target hardware. The compiler uses a number of advanced optimization techniques, such as polyhedral optimization and automatic parallelization, to generate code that is both correct and fast.

By decoupling the basic expressions of the algorithm from the scheduling and optimization strategies, Halide makes it easier for developers to write high-performance image and signal processing applications without having to worry about the low-level details of optimization. The approach used by Halide is similar to other high-level programming languages and frameworks, such as TensorFlow and PyTorch, which provide abstractions that allow developers to focus on the high-level logic of the algorithm without worrying about the low-level details of hardware optimization.

3.5.2. Tensor Virtual Machine (TVM)

TVM is a powerful tool for optimizing and deploying deep learning models on a wide range of hardware platforms. One of the key features of TVM is its ability to expose graph-level and operator-level optimizations for DNN (Deep Neural Network) workloads across diverse hardware back-ends. TVM supports a wide range of optimization techniques, including operator fusion, tiling, and memory hierarchy management. Operator fusion involves combining multiple operators into a single kernel, which can reduce memory bandwidth and improve cache utilization. Tiling involves partitioning the input and output tensors into smaller tiles, which can reduce memory latency and improve data locality. Memory hierarchy management involves managing the movement of data between different levels of memory, such as between cache and main memory. Aside from these techniques, TVM uses two representations to optimize DNN workloads: Relay and TensorIR. Relay is a high-level neural-network-layer representation that can automatically generate kernels based on this representation. At the Relay level, TVM can fuse operations to improve runtime and reduce memory overhead. TensorIR is a low-level loop-based representation that can be emitted as LLVM IR for use on GPUs or as C code for microTVM (embedded) deployments.

TVM's main optimization strategy uses the Halide functional programming language, which allows for splitting up loop-computations and matching them to multiple threads or SIMD instructions. Halide uses the equivalence principle inherent to functional programming to propose many valid computational schedules that can optimize for SIMD-instructions or hardware caches by treating the optimization of schedules. These schedules can be measured for runtime on the hardware, which can optimize the compiler output (= Autotuning). However, the search space for this autotuning problem is defined by the optimization knobs exposed in TVM's tensorIR recipes (TOPI), which requires expert knowledge for proper autotuning. The halide model also treats temporal and spatial loop splitting separately, while polyhedral optimization treats them jointly.
One limitation of the halide model is that it does not allow for side-effects, which is typical in explicitly managed scratchpad memories. An extension to Halide called Exo solves this problem by employing an SMT solver to prove equivalence and enable side-effects and thus for explicit memory transfers.
Overall, TVM's optimization techniques and support for a wide range of hardware back-ends make it a valuable tool for researchers and developers working in the field of deep learning.

3.5.3. Tensor Comprehensions

Tensor Comprehensions(TC)[2] is a framework agnostic library to automatically synthesize high-performance machine learning kernels. The compilation flow combines Halide and a Polyhedral Compiler derived from ISL and uses both HalideIR and the ISL *schedule-tree* IR. The compiler provides a collection of polyhedral compilation algorithms to perform fusion and favor multi-level parallelism and promotion to deeper levels of the memory hierarchy. TC showed that, fixing a few predefined strategies with parametric transformations and tuning knobs, can already provide great results. In that previous work, simple genetic search combined with an autotuning framework was sufficient to find good implementations in the ***non-compute bound regime***. This requires code versions obtainable by the various transformations to encompass versions that get close to the roofline limit. The ultimate goal of TC was to concretely mix Halide high-level transformations with polyhedral mid-level transformations and build a pragmatic system that could take advantage of both styles of compilation.

3.5.4. Dory

The DORY framework by Burrello et al [3] uses C library templates to drive a cluster of 8 RISC-V cores (an L2 accelerator) and coordinating tiling for optimized memory transfer. The optimized library used is XPULP-NN [4] (open-sourced here).
Tiling can in this case split up bigger layers in smaller ones that can still be executed on smaller memory accelerator devices, as long as the top-level memory hierarchy is large enough to store the entire operations operands. This tiling strategy is kernel-specific, and can use hardware specific heuristics to optimize tiling for specific hardware.
The tiling is then formulated as a constraint programming problem that is solved by OR-Tools. In the end, Dory generates C code that runs an entire neural network with explicit memory transfers that can be passed on to a specialized C compiler.

Dory's features:

- Parametrizable library and runtime
- Does static explicit scratchpad memory planning (3-level = DRAM, L2, L1)
- Supported by PULP-NN based library for PULP-CLUSTER = RISC-V based cluster
- Uses constraint programming (OR-tools to perform optimal memory planning) with memory constraints, layer-specific tiling patterns and hardware-specific

3.5.5. OpenXLA : StableHLO, XLO, IREE

The OpenXLA project is an ecosystem of ML compiler blocks. The project includes:

- StableHLO: StableHLO is an operation set for high-level operations (HLO) in ML models. Essentially, it's a portability layer between different ML frameworks and ML compilers. IT acts as a common, stable IR for ML frontends to lower to. StableHLO is based on Tensorflow's MHLO IR with additional functionality including serialization and versioning. The IR aims to provide backward and forward compatibility guarantees.
- XLA: XLA (Accelerated linear Algebra) is an open-source ML compiler for GPUs, CPUs and ML accelerators. The XLA compiler takes models from popular ML frameworks such as PyTorch, Tensorflow, and JAX and optimizes them for high-performance execution across different hardware platforms.
- IREE: IREE (Intermediate Representation Execution Environment) is an MLIR-based end-to-end compiler and runtime that lowers ML models to a unified IR that scales up to meet the needs of the datacentre and down to satisfy the constraints and special considerations of mobile and edge deployments.

3.6. Security in Compilation

Compilers convert a program from one representation to a different one while maintaining functional equivalence. In general, this equivalence is only roughly specified which enables the compiler to modify and optimize the generated code. Such optimizations usually target the execution time or the program size and are crucial to achieve reasonable or high performance in practical applications. However, the aspect of security is typically neglected. Common programming languages do not offer ways to convey security notions to the compiler which are then preserved during compilation. Even worse, security properties of handcrafted high-level code may be inadvertently lost during compilation because they conflict with the optimization goals. The most prominent security vulnerabilities introduced by compilers are timing and power side channels.

3.6.1. Side-Channel Attacks and State-of-the-Art Countermeasures

An attacker with physical access to the device can measure the power consumption of the chip and use it to deduce information about secret values such as cryptographic keys. Several attacks with varying complexities have been proposed in the literature. In a simple case, the power consumption varies based on the execution path of the program. In more complex settings, it

suffices for an instruction to require less energy for some operands than for others, and the secret values can be recovered.

Alternatively, the execution time of (or parts of) a program may also be exploited by an adversary to steal sensitive data. The canonical example is a branching instruction based on the bits of a secret value, but there are much more advanced attacks involving the memory access times for some addresses or the execution times of individual instructions.

Both of these have been placed into new prominence by the emergence of enclave technology. While enclaves, such as Intel SGX and Arm Trustzone, prevent access to secrets within the enclave, even by the operating-system kernel, side-channel attacks are considered out-of-scope for the hardware, and so maintaining secrecy becomes a software, and ultimately, full-compiler-stack problem. To exacerbate the problem further, transient execution attacks including Spectre (https://meltdownattack.com/) have dramatically extended the attack surface, not just from timing side channels within existing code, but any code that could be speculatively executed by the hardware in order to improve performance. This causes a dual challenge: not only must all paths that timing-sensitive code could execute, even speculatively, also be constant time, but also, no time-variant code paths may be allowed to access secret values, even down misspeculated paths.

The state-of-the-art countermeasure to prevent power side-channel attacks is *masking*. There exist several variants such as Boolean or arithmetic masking, which are optimized for different use cases. The underlying idea is the same: the processed values are homomorphically transformed to a masked domain (also called sharing) and the algorithm operates on the transformed data. The most common approach is to use Boolean sharing where a secret value x is split up into s shares such that $x = x_0$ ^ x_1 ^ ... ^ x_{s-1} (where ^ is the exclusive OR operator). The desired protection against power side-channel attacks is achieved by selecting s-1 shares uniformly at random. Other approaches use arithmetic sharing where the exclusive OR is replaced by additions or multiplications. However, in order to process the shared secret values, the target algorithm needs to be adapted as well. In general, masking linear functions is straightforward while masking non-linear functions is more challenging. Nevertheless, the output of the masked algorithm is transformed back to obtain the actual result. However, masking is often applied manually to software and hardware implementations. In both cases, it is indispensable to verify that the synthesizer or the compiler does not invalidate any of the security properties in order to improve performance.

Timing side channels can be evaded by adhering to the *constant time programming paradigm*. Analogously to masking, it is essential to verify that the compilation procedure retained all constant time properties. Alternatively, a DSL in combination with a designated compiler (e.g. FaCT) may be used. In such cases, the constant-time properties can be specified directly in the source code and are (formally) verified to hold for the compiled LLVM bitcode as well. However, such DSLs are constrained and not suitable for general application development. In general-purpose compilers, such annotations must be hacked into the IR by, for example, reinterpreting them as pseudo instructions with side effects (https://llvm.org/devmtg/2019-04/slides/SRC-TuanVu-Compilation_and_optimization_with_security_annotations.pdf), and properties

generated and verified in the compiler's middle end (https://github.com/lac-dcc/lif) are often broken in the backend instruction selection, such as select instructions becoming branches in LLVM's RISC-V backend. In hardware, even the latest enclave technologies do not provide innate side-channel resistance: Arm's CCA technology (https://haspworkshop.org/2021/slides/HASP-2021-Session2-Arm-CCA.pdf) gives software running inside the enclave "the tools to protect itself" but no guarantees in hardware, meaning the property is almost entirely devolved to the software stack.

Spectre adds a new dimension to the defences required: not only must code that handles sensitive data be constant-time, but no other time-variant code may access the data, even speculatively, if it is to be secure. Compiler techniques for automatically inserting barriers are currently insecure, low-performance, or both [5]. Barrier mitigations have been added to the Linux kernel by hand (https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html) to protect against kernel-specific vulnerabilities, but these are not the direct threat model for enclaves, where the vulnerability is in user code, and the approach currently used to decide where to place barriers is ad hoc and prone to error. Hardware solutions for comprehensive protection, such as Speculative Taint Tracking [6], see too high overheads for deployment. Codesign methods, that reorder instructions based on the limitations enforced in hardware [7] show promise but have not been examined against the strongest threat models [8] and still leave performance on the table relative to today's unsafe baselines.

3.6.2. Reverse Engineering

In general, **reverse engineering** refers to the process of analysing a product or system to understand its design, function, or components. In the context of artificial intelligence, reverse engineering[2] can be used to extract information about the inner workings and design of a neural network model.

One common method of reverse engineering neural networks can be the **teacher-student method**. In this approach, a large, complex neural network (the teacher) is trained on a dataset, and then a smaller, simpler neural network (the student) is trained to mimic the behaviour of the teacher network. By examining the output of the student network, an attacker can gain insights into the inner workings of the teacher network, potentially revealing proprietary information about the model's architecture or training data.

To prevent reverse engineering through the teacher-student method, one approach is to introduce noise or other markers to the output of the student network, making it more difficult for an attacker to discern useful information. Another approach is to use adversarial training, in which the student network is trained to resist attempts at reverse engineering by introducing deliberate misdirection or obfuscation.

An additional method of reverse engineering neural networks can be through the analysis of their output alone, without knowledge of the underlying architecture or training data. For example, an attacker might input a series of carefully constructed test cases to the network and analyse its responses to infer information about its internal structure or decision-making process.

To mitigate this special type of reverse engineering, one approach could be to **introduce watermarking**[3] into the output of the neural network. Watermarking involves adding a small, non-noticeable signal to the output of the network that can be used to identify the source of the output. This can deter attackers from attempting to reverse engineer the network, as the presence of the watermark can reveal the actions of the attacker.

3.6.3. Secure Compilation in CONVOLVE

Within CONVOLVE there are three different "types" of compilation. First, the applications (i.e., neural networks) need to be compiled with an utmost focus on high performance as well as low latency and power consumption. For this compilation type, side channels are not considered since they directly oppose all aforementioned optimization goals. Second, the relevant crypto-accelerators need to be devised, implemented, and ultimately compiled in a side-channel resistant manner. However, this is done only once and ahead of time. The final accelerator can then be seen as a secure enclave ("TEE") with a minimal interface. If the application needs to perform a cryptographic operation, it can be delegated to the corresponding accelerator which executes it securely. Third, the application needs to perform cryptographic or time-sensitive operations for which no (or only partial) accelerators are available, for example verifying a pin sequence or password without revealing timing information about how many characters are correct, potentially in an enclave where even the operating system should not be leaked this information. In the latter case, the application logic needs to be segregated from the cryptographic logic. The former can and should be optimized independently of any side-channel considerations. The latter must be programmed and compiled in a side-channel secure way, guaranteed end-to-end throughout the compilation flow such that properties verified in the language level cannot be violated by the compiler's middle end, and properties verified in the middle end cannot be violated by instruction selection or the choice of microarchitecture, including forms of speculation introduced by the target microarchitecture. In practice, this depends on the security goals and entails adhering to the constant time programming paradigm, masking the relevant code, verifying the object file or using a DSL with a designated compiler.

These will allow separation of duties: if a region of code has no timing-side-channel constraint applied to it, then it can be optimized to the maximum allowed. If it has a constraint in place, then code can still be optimized, but only in a way that preserves or even generates timing invariance, and only ever emits code at the backend that is guaranteed secure, since the constraint cannot be thrown away by any compiler pass. For running code, in enclaves or otherwise, while maintaining secrecy even with Spectre vulnerabilities, we will consider mechanisms for code reordering [7], potentially adding new hardware restriction mechanisms to allow high-performance defence against the strongest threat models. Our code reorderings, encoding concepts of multiple independent regions of speculation, will restore the memory-level parallelism that threatens to be destroyed unless safe techniques can be developed.

4. High-level Description of the Target Architecture and its Accelerators

We first describe the PULP (Parallel Ultra Low Power) platform and then describe the proposed accelerator interface and different possible accelerator designs.

4.1. The PULP platform

The PULP platform (project website, github) is an open-source platform between ETH Zurich and University of Bologna which develops open, scalable RISC-V based hardware and software with a focus on energy efficiency. The project has developed a variety of open-source RISC-V processor cores, peripherals and further IPs needed to design complete System-on-Chips (SoCs). In the CONVOLVE project many of the PULP IPs will be used and extended by various partners to design a heterogeneous and modular System-on-Chip (SoC) with specialized new accelerators.

The open-source project offers already many relevant hardware and software blocks and makes heavy use of the open-source instruction set architecture (ISA) RISC-V. The available IPs include a variety of RISC-V cores, from fully Linux-capable devices to low-power microcontroller cores. Within their research efforts, they have implemented a variety of RISC-V ISA extensions (XpulpV1, and XpulpV2) which reduce the overall cycle count of the program and increase overall energy efficiency – even neural network specific ones have been designed called (XpulpNN). The extended instructions include for example hardware loops, post-increment loads and stores, and packed-SIMD dot-product operations. These extensions are supported in custom LLVM backends and GCC compiler suites through automatic optimization and built-in functions.

To aid in the development of software on these platforms, PULP also has open-source simulators. The C++-based GVSoC simulator [9] supports running semi-cycle accurate simulations (up until 90% accurate) of PULP based systems at a much faster rate than typical RTL-level simulations. The (non-cycle-accurate but instruction-accurate) Rust-based Banshee simulator [10] was developed to make quick software verification possible for scaled-up manycore systems.

The PULP Platform has also developed a software stack to train, tile and deploy quantized neural networks (QNNs) onto their systems. An overview is given (Figure 2) and includes QuantLab and Dory which are discussed in Section 2.3.1 and 2.5.4 and PULP-NN, a C library with optimized neural network (NN) primitives exploiting for example the XpulpNN ISA extensions.

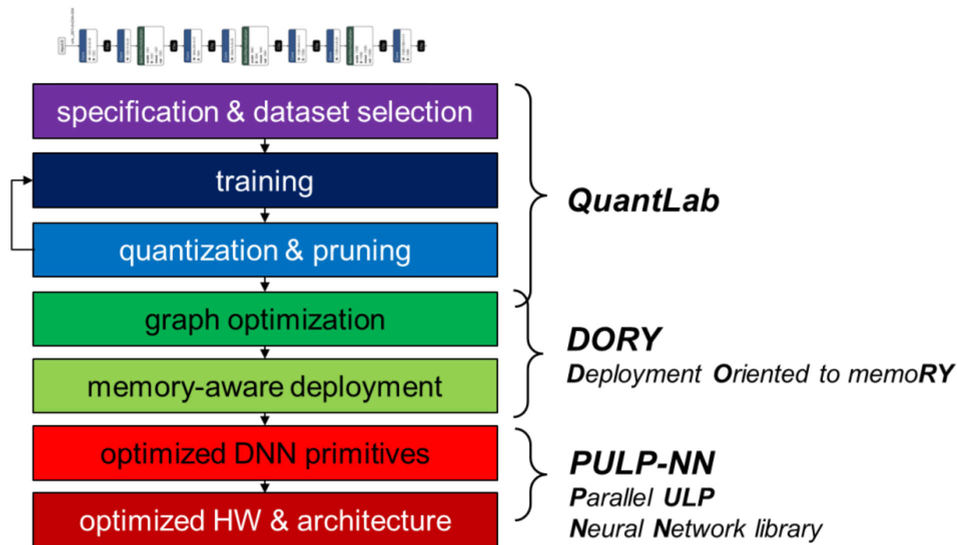

FIGURE 2. THE PULP PLATFORM SOFTWARE STACK TO TRAIN, TILE AND DEPLOY QNNS ON PULP SYSTEMS.

4.2. Programming Interface

All the accelerators proposed in CONVOLVE are accessible through different levels of memory. They are compared in the table (table 1).

TABLE 1. COMPARISON OF ACCELERATORS AT DIFFERENT MEMORY LEVELS IN CONVOLVE.

	L0 accelerator	L1 accelerator	L2 accelerator
Hardware Implementation	Accelerator which acts as a RISC-V co-processor and is integrated into a cores datapath over the core-v-xif interface.	Accelerator that shares an integrated multi-banked tightly-coupled data memory (TCDM) with RISC-V cores. An asynchronous direct memory access (DMA) unit brings in data from a main memory.	Accelerator that has its own memory and is accessed over an AXI-bus.
Programming model	Custom RISC-V extensions must be inserted by the complier.	Custom accelerator commands are issued over the respective bus (Peripheral or AXI). The L1 accelerator over memory-mapped registers or the help of a DMA unit. The L1 accelerator has streamers to fetch data from TCDM into the local buffers. The L2 accelerator potentially uses a DMA to bring data from main memory to local memory. Interfaces can be custom for each accelerator.	

4.2.1. Accelerators

Within the proposed interface many different accelerator designs in different technologies are possible. Some accelerator performance metrics can be found at https://nicsefc.ee.tsinghua.edu.cn/projects/neural-network-accelerator/ one can see that an off-the-shelf desktop CPU has a very poor energy efficiency at about 0.3415 GOP/s/W for FP16 workloads. GPUs can better exploit the inherent parallelism in NNs, yielding an energy efficiency that is up to two to three orders of magnitude better. FPGAs can specialize even better to NN workloads, but suffer from a reconfigurability overhead. ASIC designs significantly outperform all other designs, due to the absence of a reconfigurability overhead and more specialized data path, data storage, memory hierarchy and interplay between them.

Below (Table 2) are some key differentiators between several design options within the CONVOLVE project:

TABLE 2. COMPARISON OF ACCELERATORS WITHIN THE CONVOLVE PROJECT.

Accelerator Technology	Description	Intrinsic Ops	Energy eff.	Data types	Reliability
Coarse grained dense accelerators	Standard VLSI cells ASIC highly Optimized for NN operations	GEMM, Conv2D, Batchnorm, Add	High	Typically, Integer arithmetic can be Signed/unsigned. Can be scalable (int8, int4, int2, binary) *High on memory traffic*	High, fully digital
Spiking accelerators	Specialized datapath for Spiking operations	Basic weighted sum operation	Very high	Spikes. Binary spike and bfloat16 for synaptic weight	High for fully digital Implementation, Inherently resilient for nonidealities in analog implementation
CGRA	CGRA: Coarse grained reconfigu-rable array	Usually very flexible, tuneable ISA (even complex operations) reconfigurable datapath	Higher Than FPGA. Very high If very coarse grained	Depends on design, typically int8,16,32 Smaller data types by subword	High if Digital CMOS is used. Can be combined with near Vdd, Vstacking, approx units,

			operati-ons are included	parallelism (SIMD)	etc., Introducing Reliability/accur-acy issues
SRAM-based compute (Digital)	Standard SRAM macro with periphery for CIM data processing	Typically, matrix-vector multiply accesses SRAM 1-row at a time, integrate and accumulate in the periphery	High	Typically, lowbit integer types (e.g. int4). No analog effects on computation. *Very high on memory traffic*	High, fully digital
SRAM-based compute (Analog)	Adapted SRAM macro to perform computat-ions in analog domain	Typically, Matrix vector multiply access SRAM N-rows together, ADC and accumulation in the periphery	Higher than SRAM based digital	Ternary weights, Arbitrary Integer precision inputs, analog domain can induce noise in computation and requires careful quantization very high on memory traffic,	Noise sensitive (like V, T, process). Limited accuracy
RRAM based	Resistive RAM allows for nonvolatile compute near memory architectu-res	Typically, Matrix vector operation with Multiply Accumulation (MAC) operation as its kernel operation	Very high due to Non-volatility (zero leakage)	Operations are performed in the analog domain.	RRAM has some endurance Issues when performing multiple write operations. Hence, it is suitable for inference operations where the RRAMs are programmed once/less frequently.

5. Roadblocks

Despite the numerous pre-existing DNN compilation translating deep neural networks to efficiently use accelerators, it is still a difficult task that either requires industry-scale engineering to automate this process or significant manual efforts otherwise. Several

established DNN compiler frameworks are open-source and can in-theory be used for our efforts. However, they a need to be adapted to our use cases and different frameworks are suitable to such adoptions at different levels. In the following, we summarize some of the challenges that we have observed over the last years.

LLVM / GCC

While LLVM / GCC are not necessarily a DNN compiler framework C/C++, compilers are an important part of the CONVOLVE compiler stack. In particular, the L0 instruction set extensions that may be used to accelerate RISC-V cores must be supported by these compiler toolchains to enable developers to write optimized kernels in C/C++ to target such instruction set extensions. In particular, such optimized kernels are the foundation of DNN libraries such as PulpNN. While LLVM and GCC are production quality compiler stacks that offer all the necessary features to support such instruction set extensions, only a small set of expert compiler developers can add such extensions. This hinders rapid prototyping and effective design space exploration. While alternatives would be desirable, the ecosystem pressure to use LLVM/GCC at least in parts remains high and the cost of developing alternatives is likely prohibitive.

Halide / TVM

TVM is one of the leading deep learning frameworks and has very powerful tooling thanks to its auto-scheduler. However, as a successor of Halide it remains a single-domain compiler that was expanded from supporting image processing on 2D pictures to multi-dimensional tensors. While TVM supports a large set of CPU and GPU targets, extending it with custom features – especially ones that are accelerators specific – is increasingly difficult. In particular, TVM does not offer a state-of-the-art compiler design framework that makes it easy to instantiate new IR abstractions and does not offer the large amount of tooling that we are used to from compilers such as LLVM. Similarly, any extension we develop for TVM does not benefit from synergies with compilers that are not deep learning focused. Finally, TVM as a deep-learning only compiler prevents close coupling with low-level hardware lowering, e.g., to RISC-V.

Tensor Comprehensions

This project is not active anymore and its developers now work on MLIR. The reasons to move away from Tensor Comprehensions were:

- Halide was never properly used in Tensor Comprehensions beyond shape inference. Most of the investment went into simplifying polyhedral transformations and building a usable end-to-end system. MLIR was deemed a better infrastructure to mix these types of compilation.
- The early gains provided by reusing established infrastructures (HalideIR and ISL schedule trees) turned into more impedance mismatch problems than could be solved with a small tactical investment.
- Tensor Comprehensions emitted CUDA code which was then JIT compiled with NVCC from a textual representation. While this was a pragmatic short-term solution, it yielded

hard to perform low-level rewrites that would have helped with register reuse in the ***compute-bound regime***.

- The same reliance on emitting CUDA code made it difficult to create cost models. This made it artificially harder than necessary to prune out bad solutions. This resulted in excessive runtime evaluation, as reported in the paper [11].

Tensor Flow / IREE / Open XLA / JAX

The Google deep learning stack is complex and fragmented yet has a lot of promising technology. While the original TensorFlow code generation has been replaced by Google with an IREE and OpenXLA based flow, IREE, OpenXLA as well as JAX together form an integrated ecosystem that in large parts is very useful to us. While the use of MLIR within this framework facilitates interactions, the design of reusable components, and the various useful abstractions in the Google ecosystem are spread across different projects and not all of them are as mature as CONVOLVE requires. IREE has supported for a long time only some generations of CPUs. Moreover, accelerators such as GPUs and even more so CONVOLVE style L1/2 accelerators are practically not supported. Similarly, the connection of the Google ecosystem with PyTorch is rather limited.

PyTorch

PyTorch is likely the most widely used DNN compiler framework. It is incredibly well accepted by deep learning engineers and recently even gained a new compiler framework written purely in Python. Yet, PyTorch has very limited support to target custom accelerators and the underlying IRs that one can represent are not sufficiently general to express many accelerator concepts. In particular, PyTorch can only represent graph style IRs that look like deep neural networks, but has very limited support for hierarchical IRs that are well suited to model loop nests or GPU and accelerator code launches. PyTorch also has very limited tooling to effectively support compiler developers and has little intentions to become a generic compiler framework. Hence, building large amount of tooling around the PyTorch compiler stack is unlikely to result in broad use of our tools.

MLIR

MLIR as a project offers a full-fledged compiler framework, yet it only provides a couple of abstractions out of the box. It is more a tool for building compilers than a complete compiler framework. As a tool, it is perfectly useable and has incredible community support. Yet, it is entirely insufficient when one expects an out-of-the-box deep learning compiler. Even as a compiler framework, it is only easy to use for compiler experts. Non-experts, application or hardware engineers, or general developers who want to tailor a compiler to their specific applications or accelerators will require months of training to work with MLIR. As a result, it is not suitable for rapid prototyping or fast-moving research, but more for deploying production quality compilers.

Dory

ETH's in-house compiler stack is tailored to their in-house hardware and likely the closest in terms of end-to-end compilation flow that we have when aiming to target CONVOLVE style accelerators. However, Dory has very much been tailored to ETH hardware, is not very extensible, and shares with TVM the constraint that it was never intended to be very extensible. Hence, non-standard DNNs, control flow as needed for dynamic DNNs, and many other extensions are hard to integrate into Dory. Similarly, the tooling around Dory is very limited to compare to the industry-level tooling frameworks such as LLVM and MLIR offer.

Overall, we have many powerful frameworks that are excellent in certain aspects. Yet, none fully covers the needs of CONVOLVE. Many also do not offer the extensibility and code-reuse potential that we need to cover the diverse applications and accelerators in CONVOLVE.

6. Reseach

We first describe a preliminary design for the compilation pipeline. We then describe the research question of how hardware experts can interface with the pipeline to introduce expert knowledge into the compilation. We further describe dependencies on other WPs and the use cases addressed by this WP. Finally, we list out the contributions of this WP within the project.

6.1. Compilation Pipeline

The CONVOLVE compilation flow's main objective is to effectively map deep neural networks on heterogenous chips consisting of RISC-V CPU clusters and the various CONVOLVE accelerators. To maximize the scalability and impact of our compilation flow and enable it to scale to real-world workloads, we will take advantage of pre-existing efforts and make it available in a form that maximizes collaboration and reuse. Hence, we will rely on established open-source technology stacks and evolve them where needed.

The CONVOLVE compiler is divided into different parts: Frontends, Greybox Compiler, Compiler Interaction Tooling, Analysis, Optimizing & Scheduling.

Frontends

We support two main frontends: Tensorflow and PyTorch. By default, these frontends use eager mode compilation, in which a function is executed as soon as it is seen. Instead, we use JIT compilation from these frameworks in which the computation is represented as a graph, which can be further lowered into a common IR. We impose a set of requirements on the inputs to make sure that they are JIT compilable. InputIR is a common IR we use as an input to our MLIR Compiler. As of now, we use MHLO as the InputIR, but this will change as there is community effort to build a common InputIR. Potential replacements include StableHLO, or MLIR-TCP. We will consider including JAX and TOSA as frontends because they have a lowering to MHLO, but we leave this decision to be guided by our research outcome.

Greybox Compiler & Compiler Interaction Tooling

The Greybox Compiler (Figure 3) is based on the MLIR compilation framework. It receives input as InputIR, which is lowered into a mix of dialects that co-exist together. We take existing MLIR dialects like Linalg and SCF to represent the actual computation over tensors. We model the interfaces provided by the accelerators as hardware abstraction dialects which allow us to model memory transfers, accelerator control, direct accelerator calls as dialects. The tensor computations in Linalg and SCF can be lowered to LLVM, SPIRV, or direct accelerator calls. Our compiler will be a greybox compiler as the compilation flow can be changed without recompiling the compiler. This offers experts more control over the compilation flow and will be implemented using dynamic dialects such as the transform dialect, PDL, or IRDL. To make the process of changing the compiler or building domain specific compilers particularly easy, we will integrate and evolve the xDSL framework to offer Python-Native compiler interaction tooling where users, e.g., application or hardware experts can reach, interact, and evolve compilers using Python.

Analysis

The analysis hooks into the MLIR or xDSL compiler to provide information to optimize the current workflow of the compiler. This allows static analysis, such as cache analysis or profiling to influence the compilation flow for a particular input based on the input's characteristics.

Optimizing & Scheduling

Optimizing & Scheduling is the end of the compiler pipeline, where the output of the MLIR compiler is first automatically optimized and then scheduled onto the hardware accelerators. The scheduling infrastructure takes input from multiple analyses such as energy estimation, tools such as ZigZag, or application and hardware experts.

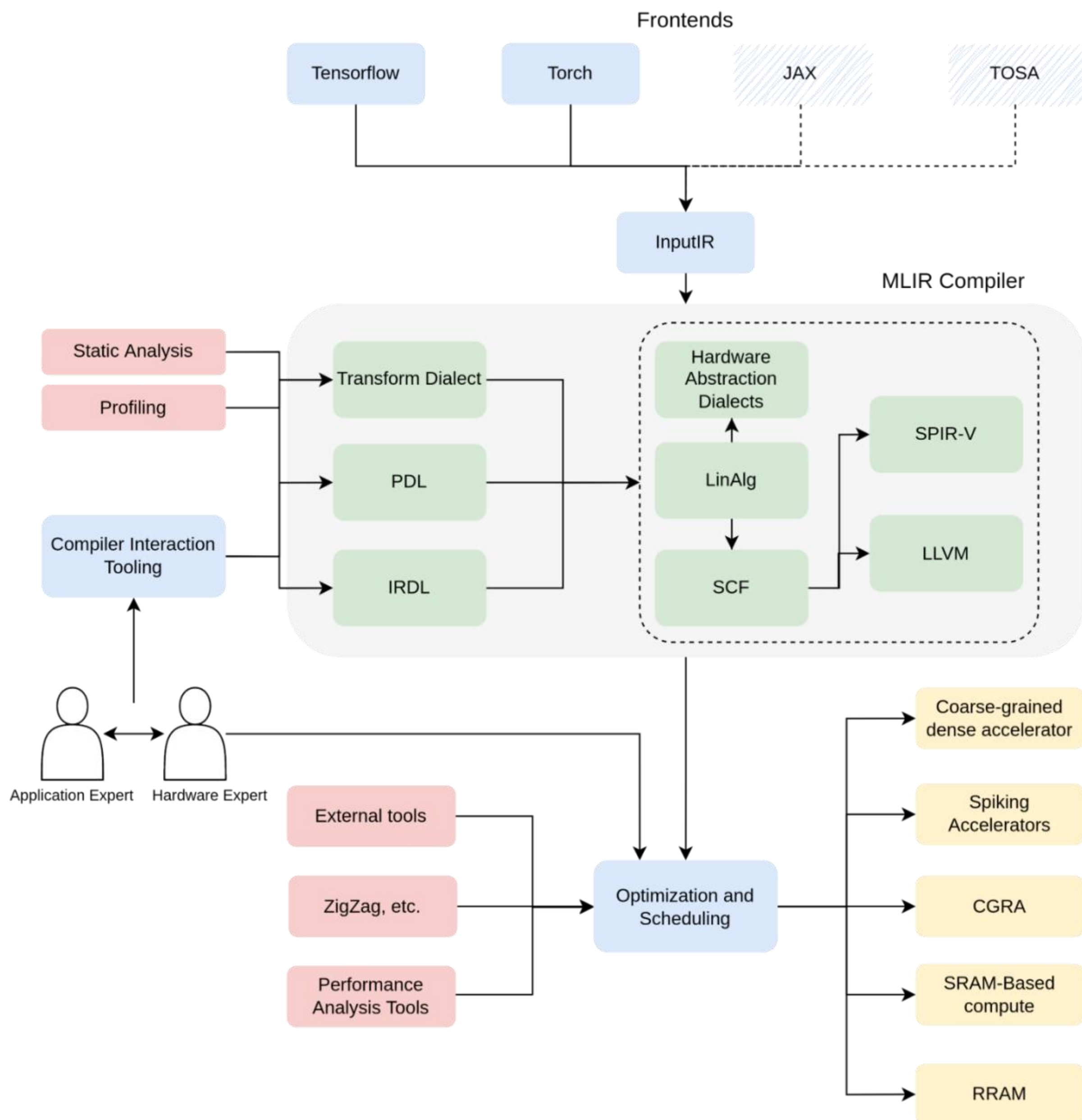

Figure 3. The Compilation Pipeline.

The above flow is a preliminary design that will be adapted based on input from the application and hardware developers as well as based on the evolution of the deep learning compiler community. In particular, we do not aim to compete with existing or upcoming deep learning compiler frameworks but will embrace change in our roadmap.

6.2. Research on Interfacing with the Accelerators

A key research question in CONVOLVE is how our compiler can best interact with accelerators and this question is particularly hard to solve. ETH Zurich and KU Leuven have both specialized tools (e.g., Dory, ZigZag) to cover part of this question. The tools are very effective but limited in scope and entirely independent of the open-source compiler stack proposed by MLIR and many industry partners. While CONVOLVE can and will rely on pre-existing tools as a first step, we aim

to research if we can integrate the ideas or tools better with the MLIR compiler stack. Yet, the degree to which this integration will take place is an open research question for us.

Our initial approach towards interfacing with accelerators through MLIR will embrace existing best practices used by our partners as well as industry. In particular, ETH uses an accelerator library called PULPNN that mirrors industry efforts, e.g., Intel's tensor processing primitives, where hardware experts write optimized kernels for a given hardware that can be used by tools such as compiler to effectively target this hardware. These kernels contain all the information needed to move data to an accelerator, run on an accelerator, and eventually move data back to host memory. From a compiler perspective these libraries look very much like a function call or a high-level intrinsic. Hence, as a first step we will investigate if accelerator support can be provided using a similar interface where hardware experts provide their optimized kernels as well as mappings from compiler-internal operations to these primitives. The compiler can then automatically select the relevant primitives.

6.3. Dependencies with other WPs

As the compiler is the connection between application and hardware stack, it is heavily interconnected with the other work packages. In particular, we require the following input from other work packages.

WP1

- The use case providers in WP1 provide us with code examples for their applications that influence and continuously inform the design of the compilation flow along multiple dimensions.
- The use case providers use our compilation flow to optimize their applications and provide feedback on the features that are needed.
- The use case providers provide optimizations at the application level and annotate their applications with information that facilitates the accelerator mapping.
- The neural networks provided must follow these requirements:
- The code given should be either in PyTorch 2.x or Tensorflow 2.x. Older versions of PyTorch and Tensorflow are not supported.
- The given code must separate what is meant to be compiled and what is supposed to run in python. The code meant to be compiled must be jit compilable in the frontends.
  - For Tensorflow, this Is done by decorating a function with tf.function(jit_compile=True) to mark it for compilation. The given function must compile after decorating it with this method.
  - For PyTorch, this is done by decorating a function with dynamo.optimize to mark it for compilation.

WP2

- A set of interface specifications at the ISA-level for L0 accelerators and a DMA-based interface definition for L1/L2 accelerators.

- A set of kernel operations for the accelerators (e.g., similar to PulpNN), where the high-level interface of the kernel has been agreed on with WP1 and where the low-level implementation of the kernel operation is hand-optimized for the specific accelerator.
- Two models for the accelerators that – at best – can interface with MLIR or are even closer integrated with MLIR.
    - A behavioural simulator.
    - A performance model.

WP3

- Information on the following types of security aspects:
    - Threat models for security guarantees.
    - Real-time guarantees.
- Test cases to evaluate security aspects.

WP4

- We do not expect major input from WP4. As our compiler pipeline will be used by WP4, general ideas and feedback on how to evolve our compiler framework, analysis, and optimizations may emerge.

WP5

- n/a (this is our work package)

WP6

- A runtime library to enable/disable execution on the accelerators, setup memory, etc.
- Pre-existing tools to orchestrate accelerator mapping (e.g., Dory) and Quantization (QuantLab)
- Collaboration on developing an MLIR-based accelerator mapping flow (with KU Leuven)

6.4. Use Cases Requirements Addressed by the WP

We aim to address the following use case requirements.

- UC Vinotion – *NF3 Tooling for rapid application development in software*
    - § We will facilitate rapid application development by embracing existing Python-based DNN workflows as frontends, offering a grey-box compiler experience in Python, and by providing a Python-based simulator.

- UC GNA - *B1 Feedback of speech quality assessment to user so the user can intervene manually* and *B2 Knob for changing the amount of (denoising) processing taking place to empower individual preferences.*

    - § By offering easier access to the compiler, individuals will be able to change the behaviour that affects compiler decisions. WP5 can support GNA efforts to integrate domain-specific decision points into the compilation flow and connect the compiler with a potential domain-specific frontend.

- UC TASE – *F1 Possible continuous integration of HW/SW improvements and new features*

  § We will include the outcome of static analyses and profiling analyses in the code to speed-up and serve future software optimizations

Security concerns are addressed indirectly through the compiler, by matching the software to the corresponding, secure, hardware platform. Potential software-hardware co-designs to address speculative side-channel attacks are also discussed in section 3.6.3.

6.4.1. Use-Case 1: Speech Quality and Denoise

6.4.1.1. Description

The uses cases identified by Jabra address the challenge of improving speech quality in telecommunication and can be divided into **Deep noise suppression** and **Speech quality prediction**.
Deep noise suppression refers to the use of deep learning algorithms to remove noise from audio signals containing speech, such as those acquired by a headset or speakerphone microphone. This task involves training a neural network to learn a mapping from a noisy signal to a clean one, typically using large amounts of synthetic data.
Speech quality prediction involves using a deep learning model to predict the perceived quality of a given noisy/degraded speech signal, without relying on a reference clean signal. The network is typically trained using a dataset of speech signals that have been subjectively rated by human listeners or scored using existing full-reference SQ metrics such as PESQ.

For deep noise suppression, the following baseline architectures have been identified: **DEMUCS**, **CRUSE**, and **NsNet2**.
DEMUCS is an encoder-decoder architecture originally developed for source separation. The encoder consists of a series of layers performing 1D dilated convolution that extract high-level features from the input waveform. The decoder comprises a series of transposed convolutional layers that estimate a clean signal. Between encoder and decoder, an LSTM recurrent network performs sequence modelling on the embedded representation. The model features skip connections between analogous encoder and decoder layers, similarly to U-Net.
CRUSE features a similar architecture, also inspired by U-Net, but operates in the time-frequency domain (i.e., on spectrograms) using 2D strided convolution and transposed convolution for encoder and decoder, respectively. Furthermore, the sequence modeling in the bottleneck is performed by a GRU layer.

Finally, NsNet2 features a sequence of fully-connected and GRU layers. These are combined so as to extract features from the input spectrogram, perform sequence modeling, and finally derive a denoising mask that can be applied to the input.
For speech quality estimation, some of the main models in the literature include **DNSMOS**, **NISQA**, and **QualityNet**.
DNSMOS is a convolutional model featuring four blocks of 2D convolution, ReLU activation, max

pooling, and dropout, followed by two dense layers. It is trained on a set of 600 noisy speech clips that have been processed through a variety of noise-suppression algorithms, whereas the target MOS were gathered through a human subjective listening test based on the ITU-T P.808 standard.

Similarly, NISQA comprises a convolutional frame-wise feature extractor, followed by a self-attention block to model temporal dependencies, and finally an attentive pooling mechanism. This model can be trained to predict both MOS and four additional quality dimensions. Finally, QualityNet use a bidirectional-LSTM block followed by fully-connected layers predicting frame-wise MOS predictions which are then averaged together to provide a global score.

Although new denoising and speech quality prediction models are constantly being developed, these models bove exemplify the use of several deep learning primitives, such as convolution, attention mechanisms, and recurrent units. The existing code for some of these models and use cases can be found here: https://gitlab.tue.nl/es/convolve/wp1-use-cases. At the moment, this includes:

- a PyTorch implementation of DEMUCS
- a TensorFlow implementation of a UNET-based model
- a PyTorch implementation of NsNet2, UNET, and CRN, with hopefully more models coming (dyn_experiments repo)
- relevant datasets for denoising tasks (gna_datasets repo)
- a collection of full-reference speech quality metrics, that can be used to evaluate denoising tasks or to generate labels for SQP.

6.4.1.2. Corresponding Security Concerns

In general, GNA is not overly concerned about overall lack of privacy or safety from the user's perspective, since no data is stored on the device, and data transmission commonly occurs within inherently insecure channels (air medium) or channels where security is ensured by the underlying transmission protocol (i.e. bluetooth or other RF).

However, one realistic concern is the **protection of intellectual property** in form of specialized data and neural network architecture which can be expensive to acquire and train. This can and has been threatened e.g. by the use of reverse engineering techniques.

In general, **reverse engineering** refers to the process of analyzing a product or system to understand its design, function, or components. In the context of artificial intelligence, reverse engineering[4] can be used to extract information about the inner workings and design of a neural network model.

One common method of reverse engineering neural networks can be the **teacher-student method**. In this approach, a large, complex neural network (the teacher) is trained on a dataset, and then a smaller, simpler neural network (the student) is trained to mimic the behaviour of the teacher network. By examining the output of the student network, an attacker can gain insights into the inner workings of the teacher network, potentially revealing proprietary information about the model's architecture or training data.

To prevent reverse engineering through the teacher-student method, one approach is to introduce noise or other markers to the output of the student network, making it more difficult for an attacker to discern useful information. Another approach is to use adversarial training, in which the student network is trained to resist attempts at reverse engineering by introducing deliberate misdirection or obfuscation.

An additional method of reverse engineering neural networks can be through the analysis of their output alone, without knowledge of the underlying architecture or training data. For example, an attacker might input a series of carefully constructed test cases to the network and analyse its responses to infer information about its internal structure or decision-making process.

To mitigate this special type of reverse engineering, one approach could be to **introduce watermarking**[5] into the output of the neural network. Watermarking involves adding a small, non-noticeable signal to the output of the network that can be used to identify the source of the output. This can deter attackers from attempting to reverse engineer the network, as the presence of the watermark can reveal the actions of the attacker.

6.4.1.3. Reference to the Complied Code

Compiler experiments: https://gitlab.tue.nl/es/convolve/wp5-compiler/kunwar-experiments

6.4.1.4. Characterization of the Use Case

In order to analyse the performance of the Demucs neural network, profiling data was collected to identify areas that could be optimized to improve the overall execution time.

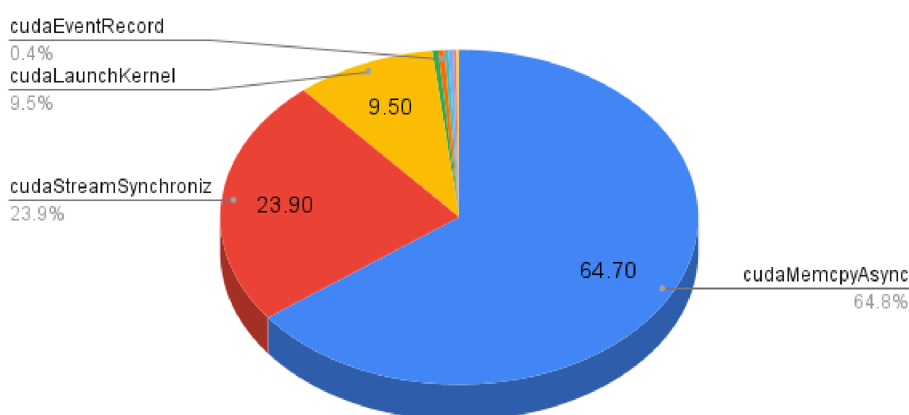

FIGURE 4. MEMORY TRANSFERS ACCOUNT FOR 2/3 OF THE EXECUTION TIME

The profiling data (Figure 4) shows that a significant portion of the time is spent on data transfer operations, specifically the **cudaMemcpyAsync** operation, which accounted for 64.7% of the total

time. This indicates that there is a need to optimize data transfers in order to reduce cache misses and improve the overall performance of the neural network.

To address this issue, several strategies are suggested, including restructuring the data to reduce cache misses, fusing operators together to increase data reuse, and adding prefetch logic to eliminate cache misses. These strategies are aimed at optimizing the data transfer process to minimize the impact on the overall execution time of the neural network.

Another area that is identified for optimization is the **cudaStreamSynchronize** operation, which causes the host to stall until the stream completes all its tasks. This accounts for 23.9% of the total time and indicates that there is a need to overlap computation and communication to reduce the overall execution time.

To address this issue, multiple streams are suggested to perform computation and data transfer concurrently, which would significantly improve the performance of the neural network. By overlapping computation and communication, the overall execution time of the Demucs neural network can be greatly reduced, making it a more efficient and effective tool for audio source separation.

The profiling data also revealed that a significant portion of the time is spent on specific kernels within he Demucs neural network. In particular, the **cutlass** kernel accounts for 17.3% of the total time, while the **xmma_cudnn** kernel accounts for 15.6% of the total time.
This indicates that by identifying the bottlenecks and optimizing these kernels, the overall performance and energy-efficiency of the Demucs neural network can significantly improve.

6.4.2. Use-Case 2: Audio Detection and Tracking

6.4.2.1. Description

This use-case focuses on the analysis of acoustic scenes. More precisely, we consider typical traffic scenes as recorded by cars equipped with arrays of microphones. These signals are complex superpositions of non-stationary sound sources like cars, emergency vehicles as well as stationary emitters like people taking, construction site noises and many more. The aim is to reconstruct information about the individual acoustic sources that could either augment existing information obtained from other sensors or provide additional, safety-critical information.

Here, we dissect the analysis of traffic scenes based on acoustic features into two parts: First, siren sounds should be detected and, second, the position of these sources should be reconstructed based on the signal recorded by the set of microphones. For that purpose, we draw on recurrent neural architectures that are trained on features extracted from the raw audio signals to predict the presence of an emergency vehicle and in a second step its position. The models feature a recurrently connected hidden layer composed of either GRUs or LSTMs and dedicated readout layers. The latter contains linear units with activation functions determined by the predicted observable. For the detection, a set of linear units with sigmoidal activation function is trained to output a signal as soon as a sound source of a specific kind is present within

the scene. For the tracking a set of readout modules is tasked to output spatial information for a set of sound sources. Each module features three units to predict the distance as well as angular information for a single sound source.

6.4.2.2. Conclusion

The strategies for optimization described in the paper can be utilized for audio detection and tracking for object detection and classification. Audio processing also involves the use of deep neural networks which need considerable memory for storing intermediate feature maps. By minimizing the number of memory accesses required for these intermediate feature maps using techniques such as data reuse, operator fusion, and prefetching data, the performance of audio processing can be improved substantially. This can lead to greater energy efficiency, quicker execution time for audio detection and tracking operations.

6.4.3. Use-Case 3: Image Processing

6.4.3.1. Description

For image processing, one of the most popular deep learning algorithms is YOLO (You Only Look Once) which is used for object detection and localization in images. YOLO is capable of detecting multiple objects in real-time with high accuracy and has a wide range of applications, some of which include:

1. Autonomous driving: YOLO can be used to detect other vehicles, pedestrians, traffic signs, and road markings, which is important for developing autonomous driving systems.
2. Security and surveillance: YOLO can be used to detect and track people, vehicles, and other objects in real-time, which is useful for security and surveillance applications.
3. Security and surveillance: YOLO can be used to detect and track people, vehicles, and other objects in real-time, which is useful for security and surveillance applications.
4. Healthcare: YOLO can be used to detect and analyse medical images, such as X-rays and MRIs, to help diagnose diseases and monitor treatment progress.
5. Environmental monitoring: YOLO can be used to detect and track wildlife, vegetation, and other objects in natural environments, which is important for environmental monitoring and conservation efforts.

6.4.3.2. Characterization of the Use Case

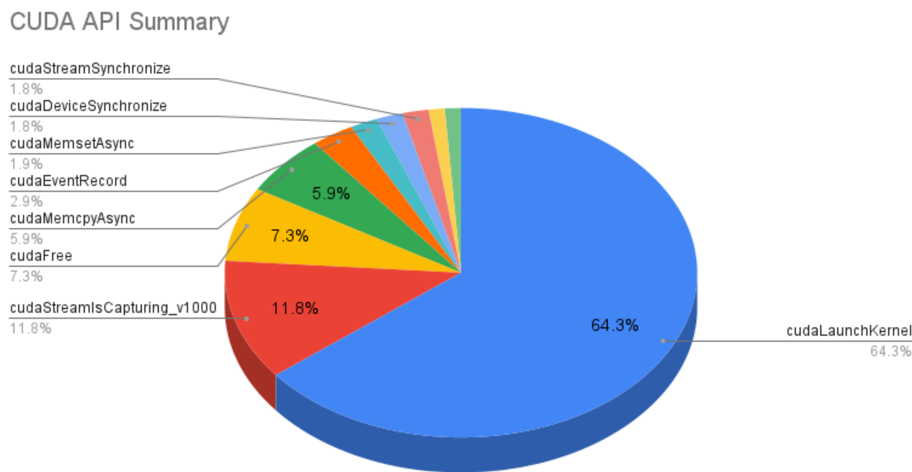

FIGURE 5. CUDA API SUMMARY YOLO_V7.

Based on the profiling results, it seems that the majority of the time (64.3%) was spent in the **cudaLaunchKernel** function, which is responsible for launching CUDA kernels on the GPU. This indicates that the program is heavily reliant on GPU computations and that optimizing the kernel execution could lead to significant performance improvements.

In the specific context of optimizing the **cudaLaunchKernel** API, memory dependence analysis and flow analysis can help identify opportunities to optimize the data access patterns and control flow of the kernel function being launched. By optimizing the kernel function, the **cudaLaunchKernel** API can be used more efficiently, which can help reduce the overall execution time.

Cuda Kernel summary (table 3) provides us the information about total execution time, Number of times kernel was called, average time for the kernel execution, minimum and maximum time taken to execute a kernel.

** CUDA GPU Kernel Summary (cuda_gpu_kern_sum):

TABLE 3. KERNEL SUMMARY.

Time (%)	Total Time(ns)	Instances	Kernel	Avg(ns)
15.5	1,32,19,64,293	2,33,303	nchwToNhwcKernel	5666.3
11.0	93,67,95,389	2,19,552	elementwiseKernel	4266.8
8.5	72,28,50,355	39,255	cutlass_tensorop	18,414.2
7.7	66,04,22,481	2,05,590	Vectorized_elementwise	3,212.3

As we can see (Table 3) the majority of time in spent in Kernel execution, Kernels mentioned in the table can be optimized in order to reduce the overall execution time.

6.4.4. Use-case 4: Satellite Image Segment

6.4.4.1. Description

One potential use-case of satellite imaging is object detection and segmentation, which can be achieved using the Mask R-CNN framework. Mask R-CNN extends the Faster R-CNN architecture by adding a branch for predicting segmentation masks on each region of interest (RoI) in parallel with the existing branch for classification and bounding box regression. The mask branch is a small fully convolutional network (FCN) applied to each RoI, predicting a segmentation mask in a pixel-to-pixel manner. This allows for the detection and segmentation of multiple objects within an image.

Satellite imaging can be used for various applications such as land use and land cover mapping, urban planning, disaster management, environmental monitoring, and agriculture. For example, satellite imagery can be used to detect and segment different land cover types such as forests, agricultural land, water bodies, and urban areas. This can provide valuable information for environmental monitoring, land-use planning, and natural resource management.

Additionally, R-CNN-based algorithms can be used for change detection analysis in satellite images, which involves comparing two or more images taken at different times to identify changes in land cover, infrastructure, or other features. The use of deep learning algorithms such as Mask R-CNN can improve the accuracy and efficiency of such analyses.

6.4.4.2. Conclusions

The optimization techniques discussed for the two use cases analysed in depth (Demucs and YOLO) can also be applied to the task of satellite imaging, specifically in the context of the region-based convolutional neural network (R-CNN), and Object detection and classification.

R-CNN is an important application area for DNNs, as satellite imagery analysis is used for a wide range of tasks, from mapping and monitoring to natural disaster response. However, satellite imaging datasets are often extremely large and memory-intensive, making it difficult to run DNNs on small devices or with limited resources. By applying the operator fusion, instruction fusion, prefetching data techniques, etc the memory usage and energy consumption of R-CNNs can be significantly reduced, leading to improved performance and energy efficiency.

6.5. Description of the Contributions to the Demos

WP 5 will offer support to all CONVOLVE partners with the use of existing DNN flows as well as our Grey-box compiler extensions and static analyses to enable our application and hardware experts to map our demonstrator applications to our accelerators. WP 5 will also provide automatic tooling to apply some yet-to-be-defined optimizations automatically. We will make all our compilation tools available. Either via open-source snapshots of our work or preferably as contributions to existing DNN compiler stacks.

7. Plans

The main goal is to manage and evolve the existing global deep learning compiler ecosystem and enable its effective use for the CONVOLVE applications and accelerators. We will support our machine learning and application partners to be productive in obtaining effective code for the dynamic neural networks and other novel machine learning architectures they develop. To benefit from the work in CONVOLVE, it is central for this code to benefit from the acceleration capabilities provided by the chips developed by our hardware partners. Another goal is to provide a code analysis toolkit that will provide useful estimates of hardware utilisation for developers working on optimising their neural networks and other workflows. We will approach both these goals iteratively, first setting up an environment that embraces the pre-existing manual workflows, global deep learning system, and partner-specific tools. We will then work with our partners to increase their productivity in mapping applications to accelerators. In particular, we will offer developer tooling to support the interaction with pre-existing compilation stacks, provide static analysis to increase code understanding, and automate certain program optimizations. As a result, we aim to work with our partners to deliver expert-level performance at increased developer productivity.

Our machine learning partners will need analysis tools to choose and optimise their architectures more effectively. Memory and control dependences between instructions, statements, operators etc., are crucial static analyses at the foundation of all optimizations. We will investigate techniques to efficiently include the outcome of the analyses in the code to be reused when re-compiled for new targets. These can also be used as hints to the new (co-designed) hardware to enable runtime optimizations based on static guarantees. Reusing the outcome of the analyses across different compilations, reduces analysis-time, optimization-time, and compilation-time in the future, for emerging targets.
Hence, we will provide the following static analyses and performance modelling tools:

1. Profiling, static analysis of the use cases, outcome embedded in the code (e.g., at the IR level or through code annotations). UMU
    a. Memory dependences
    b. Control-flow dependences
    c. Data sharing and synchronization
2. A static analysis to model the cache behavior of deep neural networks. EDU
3. An emulator for RISCV instructions that can be easily extended by our hardware partners to model new L1/2/3 accelerators. EDU

We will also provide the tools needed to support mapping and optimisation of the neural networks for the target architectures. Based on the analysis of the use case 1 (Speech quality and denoise), we identified that one of the main bottlenecks consists in the data transfers between the host and the accelerator. This can be addressed through several techniques, of which we are going to focus on neural networks operator fusion and data analysis to transfer data more efficiently from host to accelerators. Finally, to leverage the accelerators, we will perform static instruction reordering to enable instruction fusion.

4. We setup a first prototypical end-to-end compilation flow targeting CPUs and potentially GPUs by leveraging existing DNN compilers such as PyTorch, xDSL, and MLIR. *EDU, UMU*
5. We will facilitate the modification of and interaction with this prototypical compilation flow, e.g., by enabling easy access into this flow through Python and by lowering the barrier for introducing new abstractions and optimizations using declarative IR and rewrite definitions such as IRDL and PDL. We will develop these innovations to enable our partners to define their own abstraction and optimizations as part of a grey-box compilation flow that combines automatic and manual efforts. EDU
6. With that compiler infrastructure in place, we will extend it with further optimisations.
    a. Mapping of kernels (code regions) to CPU vs accelerators based on their characteristics *UMU, EDI*
        i. Leverage the static analyses from step 1 and integrate the mapping suggestions provided by external tools, such as ZigZag , etc.
    b. Optimizations: Fusion *UMU*
        i. On the high-level, neural networks operator fusion
        ii. On the low-level, dedicated instruction scheduling enables accelerator-specific instruction fusion
        iii. On the low-level instruction re-ordering (scheduling) can reduce speculation time and increase runtime memory level parallelism, through software-hardware co-designs
    c. Optimizations: Data pre-fetching *UMU*

Finally, it is important for WP5 and CONVOLVE to remain agile and up-to-date with latest innovations in the space of deep learning compilation. The landscape in this space is moving fast and we can maximize value for CONVOLVE by embracing and collaborating with the leading communities in this area. Hence, WP5 will closely follow the latest innovations in deep learning compilation, always ensuring that we maximize benefit from access to the latest compiler technology while considering the needs and pre-existing technology stacks.

8. Reference

1. Spallanzani, Matteo & Cavigelli, Lukas & Leonardi, Gian & Bertogna, Marko & Benini, Luca. (2019). Additive Noise Annealing and Approximation Properties of Quantized Neural Networks.
2. Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, & Albert Cohen. (2018). Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions.
3. A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi and F. Conti, "DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs," in IEEE Transactions on Computers, vol. 70, no. 8, pp. 1253-1268, 1 Aug. 2021, doi: 10.1109/TC.2021.3066883.
4. A. Garofalo, G. Tagliavini, F. Conti, L. Benini and D. Rossi, "XpulpNN: Enabling Energy Efficient and Flexible Inference of Quantized Neural Networks on RISC-V Based IoT End Nodes," in IEEE Transactions on Emerging Topics in Computing, vol. 9, no. 3, pp. 1489-1505, 1 July-Sept. 2021, doi: 10.1109/TETC.2021.3072337.
5. Marco Patrignani, & Marco Guarnieri. (2021). Exorcising Spectres with Secure Compilers.

6. J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas and C. W. Fletcher, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in IEEE Micro, vol. 40, no. 3, pp. 81-90, 1 May-June 2020, doi: 10.1109/MM.2020.2985359.
7. Tran, K.A., Sakalis, C., Själander, M., Ros, A., Kaxiras, S., & Jimborean, A. (2020). Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design. In Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (pp. 241–254). Association for Computing Machinery.
8. Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, & Alaa Alameldeen. (2021). Speculative Interference Attacks: Breaking Invisible Speculation Schemes.
9. N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini and D. Rossi, "GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors," 2021 IEEE 39th International Conference on Computer Design (ICCD), Storrs, CT, USA, 2021, pp. 409-416, doi: 10.1109/ICCD53106.2021.00071.
10. S. Riedel, F. Schuiki, P. Scheffler, F. Zaruba and L. Benini, "Banshee: A Fast LLVM-Based RISC-V Binary Translator," 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), Munich, Germany, 2021, pp. 1-9, doi: 10.1109/ICCAD51958.2021.9643546.
11. Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19). Association for Computing Machinery, New York, NY, USA, 177–183. Doi:10.1145/3317550.3321441