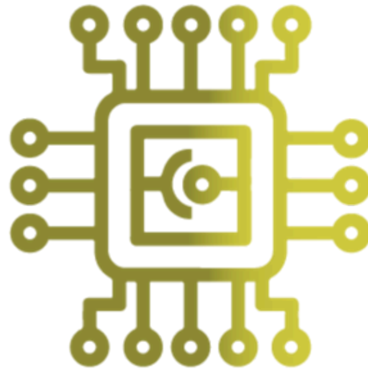


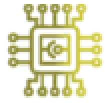
Seamless Design of Smart Edge Processors

Project funded by the European Commission

under the grant agreement number 101070374

Deliverable 5.3

Static Analysis Tools



Title of deliverable	Static Analysis Tools
WP contributing to the deliverable	WP5
Task contributing to the deliverable	Task 5.3
Dissemination Level	PU - Public
Due submission date	M33
Authors	Shreya Alladi, Emily Sillars, Ravikiran Reddy, Alessandro Cerioli, Andrea Nardi, Chistos Vassiladiotis, Tobias Grosser, Marian Verhelst
Internal reviewers	

Document Version	Date	Change
V0.1	05/03/2025	Initial Version
V0.2	13/09/2025	First complete draft
V0.3	30/09/2025	Final Version

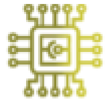
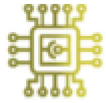


Table of Contents

Executive Summary **5**
1 Introduction and Objectives **8**
2 Background and Related Work **9**
2.1 Convolve use cases and other applications 10
2.2 Frameworks for NNs 11
2.2.1 PyTorch 11
2.2.2 ONNX 11
2.2.3 ONNX Runtime 11
2.3 Compilation Frameworks 11
2.3.1 LLVM 12
2.3.2 MLIR 12
2.3.3 IREE: An ML Compiler and Runtime Framework 12
2.4 Static Analyses 12
2.4.1 Design Space Exploration 13
2.5 Roofline Model 13
2.6 Static transformations 13
2.6.1 Tiling 14
2.6.2 Vectorization 14
2.6.3 Static Instruction scheduling for fusion 14
2.7 Quantization and Mixed-Precision Quantization 14
2.8 Memory Allocation 15
3 Static Analysis Tools and Optimizations **16**
3.1 Analysis of optimizations offered by State-of-the-Art frameworks when applied to the Convolve use cases 17
3.2 Auto-vectorization 18
Auto-vectorization is an optimization technique that automatically converts scalar operations within loops into vector operations, enabling parallel execution. This process is key to improving performance, especially for loop-heavy tasks such as those found in neural networks 18
3.2.1 Study of the factors that prohibit vectorization at LLVM IR 18
3.2.2 Study of the factors that prohibit vectorization at MLIR 18
3.2.3 State–of-the-art: ONNX-MLIR with manually vectorized libraries 19
3.2.4 Our proposal: OML-vect 20
3.2.5 Contribution: Data layout optimization for vectorization 20
3.2.6 Limits of the reduction detection in MLIR and our solution 22
3.3 Automatic Tiling 22
3.3.1 State-of-the-art in tiling 22
3.3.2 Tiling in Quidditch 23
3.3.2.1 Quidditch Dispatches 23
3.3.2.2 Tiling Dispatches 24
3.3.2.3 Double Buffering L1 Tiles 24
3.3.3 Proof of Concept: Manual ZigZag-Compiler Integration (UMU) 25

3.3.4 ZigZag-Compiler Integration....25
3.3.4.1 Compiler Targeting of Stream-Oriented Accelerators....26
3.3.4.2 Leveraging ZigZag for Design-Space Exploration....26
3.3.4.3 Automated Tiling in Compiler Middleware....27
3.3.4.4 Managing Memory Reuse and Allocation in Tiled Execution....28
3.3.4.5 Double Buffering to Overlap Memory and Compute....28
3.3.4.6 Evaluation and Deployment Benefits....29
3.3.4.7 Outlook towards fully integrated tool....30
3.3.5 Myrtle: Automatic Tiling for the Snitch Cluster....31
3.3.5.1 Tile Size Generation....32
3.3.5.2 Tile Size Selection....32
3.3.5.3 Sensitivity Analysis....32
Timing Entire Quidditch Dispatches (holistic runs)....32
3.3.5.4 Analytical - Empirical Hybrid Model....35
3.4 SigSag....35
3.5 Instruction Fusion....36
3.5.1 State-of-the-art....36
3.5.1.1 ISA-Level Fusion....36
3.5.1.2 Exploiting the Cache Access Width with Non-Contiguous Accesses:....36
3.5.1.3 Speculative fusion for non-consecutive accesses....36
3.5.1.4 Limitation of the State of the Art in Fusion:....37
3.5.2 Instruction reordering at LLVM IR for Instruction fusion....37
3.5.2.1 Validity of Reordering....38
3.5.2.3 Increasing the Compiler's Reach....39
3.5.2.4 Chained Alias Analysis using a database to reduce compilation-time....39
3.5.2.5 Limitation....40
3.5.3 Backend Analyses and Transformations....40
3.5.3.1 Contribution: Extension of the DAGMutation....40
3.5.3.2 Drawbacks of DAGMutation....41
3.5.3.3 Pseudo Instruction....41
3.6 Periodic Scheduling of Layer-Fused CNN For A Compact Schedule Representation....42
Layer-Fusion as an approach of reducing the energy intensive communication with off-chip....42
Example Network....42
Example Architecture and Motivation of Layer-Fusion....44
Line-Based Layer-Fusion Execution....44
Compact Representation of Line-Based Layer-Fusion Schedules....46
Comparison with SoTA....47
Conclusion and Future Work....48
3.7 Support for semi-automatic mixed-precision quantization....48
3.7 Integration of external libraries with the RISC-V lib ORT-SMD....49
3.2.5 Enhanced roofline model including configuration time (ACCFG)....49
4 Evaluation and Results....50



4.1 Baseline performance of neural networks compiled for x86 and RISC-V respectively.. 50
4.2 Performance analysis of NsNet....51
4.3 Performance results with and without the manual opts for vectorization.... 52
4.4 Performance results with Quidditch....53
4.5 Performance results on Instruction reordering and fusion....56
4.6 Performance results on Instruction reordering and fusion for general purpose benchmarks....57
4.7 Performance results of integrating external libraries....59
4.7.1 Optimizations for NNs for RISC-V....59
4.7.2 Quantization.... 59
4.8 Performance results with Quidditch....60
4.8.1 Cost Model Driven Tiling.... 60
5 Activities within CONVOLVE....62
6 Conclusions....63
7 References....64

CONVOLVE

Executive Summary

Our work on static analysis strengthens CONVOLVE's mission to optimize edge AI processors through innovative compilation techniques that connect general-purpose software with ultra-low-power (ULP) accelerator architectures. These static analysis methods enable automatic transformation of high-level code into hardware-optimized implementations, directly supporting CONVOLVE's core objectives of 100× energy efficiency improvements and 10x design time reductions.

We deliver **static analysis and transformation tools** that enable:

- Identifying coarse-grained opportunities to offload computations to fixed-function accelerators (UMU).
  - Identifying Snitch compatible kernels within a larger Neural Network and offloading them appropriately to the Snitch Cluster. (CAM, UMU) (3.3.2 and 3.3.5, 5.4 Integrated Tools, section 2.5.1.3)
  - Specifically: Identify and offload specialized kernels within a larger Neural Network for integration with hand-optimized library calls (e.g., vectorized GEMM, convolution) while also enabling mixed-precision quantization. (UMU, GN)(Section 3.6 and 3.7)
- Remapping data layouts at compile time to enable efficient auto-vectorization (UMU) (see section 3.2.1.5 Data layout optimization for vectorization).
- Identifying complex ternary reductions through systematic decomposition to enable vectorization (UMU)(see section 3.2.1.6 Reduction in MLIR).
- Remapping data layouts to minimize transfers and match accelerator-specific formats (UMU).
  - Manual integration of the ZigZag DSE tool's tiling recommendations for execution on the Snitch Cluster (UMU)(see section 3.3.3 PoC)
  - Full integration of ZigZag into an MLIR-based compiler to target systolic array-based accelerators (KUL) (see section 3.3.4 Praxis)
  - Automatic tiling and loop interchange for execution on the Snitch Cluster, optimized for L1 cache and register reuse. (UMU) (see section 3.3.5 Myrtle)
- Reorganizing code to alleviate performance bottlenecks and improve real-time guarantees (UMU).
  - Scheduling Instruction to facilitate fusion, which maps efficiently to the core pipeline, hence improving performance, energy and resource utilization (UMU) (see section 3.5.3 Backend Analyses and Transformations)
- Test-driven generation of optimization opportunities, dynamically verified for correctness (UMU).
  - Sensitivity Analysis to reveal trends in optimal tile size (UMU) (Section 3.3.5.3)
- Validating the correctness of compile-time transformations, identifying hot functions
- Compile time support to analyze and reorder instructions in general purpose languages, which is significantly more challenging than in compiler-friendly neural network kernels (UMU). (see section 3.5.2 Instruction reordering at LLVM IR for Instruction fusion)

- Accurate and interprocedural static analysis for AI and general-purpose code, reduced compilation time (3.5.2.4 Chained Alias Analysis using a database to reduce compilation-time).
- A performance analysis tool designed for evaluating spiking neural network architectures and hardware mappings (TUE) (Section 3.4)
- Periodic Scheduling of Layer-Fused CNN For A Compact Schedule Representation (TUE) (Section 3.4)

By automating hardware-aware code transformations, we enable:

- **Efficient accelerator targeting,** which reduces manual optimization effort.
- **Lower energy consumption** via optimized memory access and computation offloading.
- **Enhanced real-time predictability** through bandwidth-aware code restructuring.
- **Lower memory footprint** and data transfers via optimized data layouts and computation that maximizes reuse.

We strengthen CONVOLVE's toolchain by **automating critical static analyses**, ensuring high-level code can be efficiently mapped to ULP accelerators while meeting performance and energy goals.

CONVOLVE

Key Publications

- Lopoukhine A, Ficarelli F, Vasiladiotis C, Lydike A, Van Delm J, Dutilleul A, et al. **A Multi-level Compiler Backend for Accelerated Micro-kernels Targeting RISC-V ISA Extensions**. In: Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization [Internet]. New York, NY, USA: Association for Computing Machinery; 2025 [cited 2025 Mar 26]. p. 163–78. (CGO '25). Available from: https://dl.acm.org/doi/10.1145/3696443.3708952
- Alladi S, Ravindranath R, Ros A, Jimborean A. **Insights into Interpreters, Compilers, and Optimizers for Neural Networks**. In Proceedings of Workshop on Compilers, Deployment, and Tooling for Edge AI 2025. New York, NY, USA: Association for Computing Machinery;2025.(CODAI '25). Available from: (pending)
- S. Singh, A. Perais, A. Jimborean and A. Ros, "**Alternate Path μ-op Cache Prefetching**" 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), Buenos Aires, Argentina, 2024, pp. 1230-1245, doi: 10.1109/ISCA59077.2024.00092.
- S. Singh, J. Feliu, M. E. Acacio, A. Jimborean and A. Ros, "**CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions**" 2023. 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria, 2023, pp. 1-13, doi: 10.1109/PACT58117.2023.00009.
- Alladi S, Ros A, Jimborean A. **Enabling Automatic Vectorization for Transformers**. (Under review)
- Ravindranath R, S. Singh, A. Perais, Ros A, Jimborean A. **Compiler Assisted Instruction fusion** (Under review)

1 Introduction and Objectives

Task T5.4: Static analysis required to transform general-purpose code to ULP accelerators – M07 to M33 (**UMU,** UED, TASE). Developing static analyses and transformations of generic high-level code into code that **better targets hardware resources**, and identifying coarse-grained *opportunities* for targeting fixed-function accelerators via algorithmic matching. Such analyses enable tasks like **re-mapping data in memory** (e.g., data transfer reduction), targeting fixed accelerator data formats, or even **reorganizing code layout** to alleviate bandwidth bottlenecks (which will aid in both targeting custom hardware and in eliciting real-time *constraint* guarantees). Test-oriented generation of opportunities, **verified dynamically or with profile guidance**, will be used to simplify this targeting.

D5.3 Static analysis tools – Report and code of static analysis techniques for **general-purpose languages**.

As a first step, at UMU we built an end-to-end compilation framework (OML) for PyTorch/ONNX models, targeting RISC-V. This step was crucial, because state-of-the-art frameworks offered compilation support and optimizations for x86 architectures, **but not for RISC-V**. Moreover, by building our compilation flow on top of solid and widely used ecosystems (MLIR and LLVM), we not only draw direct benefits of all the automatic optimizations developed by the community, but also provided the means to extend these frameworks to add our own optimizations, tailored to the Convolve use-cases and accelerators. As a second step, we have developed static analyses both for (1) neural networks (Convolve use cases) and transformers, described in high-level languages and representations, such as PyTorch and ONNX, as well as for (2) general-purpose languages, i.e. C++. These static analyses were used to drive the Convolve compile-time optimizations in order to better leverage the underlying hardware resources.

Based on a careful analysis of the algorithms used in the neural networks and transformers, we discovered that the data layout caused poor locality and unnecessary additional data movement. To address this issue, at UMU, we performed a **re-mapping of data in memory**, by automatically transposing the matrices, which increased performance by 92% on average, by reducing the number of required memory accesses. This new data layout also made the algorithms amenable to vectorization, a highly powerful optimization, which – coupled with the **complex analysis for reduction detection** we developed at UMU – enabled **automatic vectorization** and brought performance benefits of 92% on average and up to 99%, surpassing manually tuned libraries for vectorizing. Furthermore, to get the best of both worlds, we added **support to integrate the external libraries**, e.g. manually developed in industry, and expose them in MLIR and LLVM, such that we can execute our automatic optimizations on top of the manually optimized code. In a similar vein, UMU **added an MLIR Execution Provider to the ONNX Runtime**, such that, at MLIR-level, we can benefit and build on top of the high-level graph optimizations already available in ONNX.

Starting with the NsNet use-case provided by Jabra (GNU), UMU performed **both a data layout and a code transformation, enabling automatic tiling in MLIR** for matrix-vector multiplication (present in NsNet). To this end, UMU joined forces with CAM/EDI and integrated the support for automatic tiling in Quidditch, the compiler back-end developed at CAM, targeting the Convolve hardware, Snitch. Furthermore, UMU built Myrtle, **a cost-model to drive tile-size selection**, since the size is **essential for matching the hardware characteristics**. UMU drew inspiration from ZigZag (KUL), a tool for automatic scheduling evaluation, which proposes loop interchange and tiling, but without actually generating transformed code.

Closer to the hardware, **UMU developed code layout transformations, i.e. instruction reordering**, that is initiated in the compiler middle-end and preserved and finally adapted in the **compiler back-end to match precisely the hardware requirements**. Instruction reordering enabled hardware instruction fusion on RISC-V platforms, which brings 16.6% performance benefits for neural network applications, when targeting edge devices. While neural networks exhibit a linear behavior and are more amenable to static analysis, at UMU, we took the challenge to **analyze general-purpose applications written in C++**, which abound in pointers, complex control-flow and call graphs. This required highly complex pointer analysis, inter-procedural and whole-program analysis. We extended state-of-the-art proposals and leveraged them in the context of Convolve, being able to perform code layout optimizations on complex sets of benchmarks, such as SPEC CPU 2017. We focused in particular on the representative regions of code, identified **with profile guidance**. All our transformations were performed based on the outcome of the static analysis, which guaranteed their validity. Moreover, we verified empirically the results to iron out any potential bugs.

UMU collaborated with DTU on building a library for automatic mixed-quantization, with the potential of integrating it in the OML compilation flow or for using it as an external library.

2 Background and Related Work

The section begins by presenting practical use cases that demonstrate the application of static analysis and optimization tools for neural networks (NNs). The use cases are part of WP1, provided by our partners. Additionally, we considered other benchmarks (neural networks, transformers) that embed the new features and resemble the trends in these applications. It then introduces key frameworks used in NN development and execution, including PyTorch for deep learning model construction, ONNX as an interoperable model exchange format, and ONNX Runtime for efficient cross-platform inference. The section explores essential compilation frameworks, starting with LLVM as a foundational infrastructure, MLIR for multi-level intermediate representations that facilitate advanced optimizations, and IREE, which targets high-performance execution on diverse backend architectures.
Following this, the section delves into static analysis techniques crucial for understanding and optimizing NN workloads. It highlights design space exploration strategies that systematically evaluate different implementation choices. The Roofline Model is used to visualize and analyze computational performance relative to hardware limits. Static transformations are examined, focusing on optimizations like instruction scheduling for operation fusion, which improve runtime efficiency. The section also covers quantization and mixed-precision techniques to

reduce model footprint and increase inference speed, as well as memory allocation strategies to further optimize resource use across various NNs and deployment platforms.

2.1 Convolve use cases and other applications

The proposed compiler pipeline has been developed to support a broad range of use cases outlined in WP1 by our partners (Jabra, Vinotion, Thales), ensuring coverage of all representative use cases from each contributing partner. As part of the evaluation process, we have compiled and verified the correctness of NsNet2, ConvFSNet, and DEMUCS provided by Jabra; YOLOv6 from Vinotion; YOLOv7; and an ongoing band-selection use case from Thales. Additionally, dynamic variants of NsNet2 and ConvFSNet from Jabra have been tested. The compilation flow has also been validated for transformer models, including BERT, OPT, and Whisper, demonstrating the pipeline's applicability to a wide range of modern deep learning workloads.

The table 2.1 summarizes the use cases tested.

Use Cases	Kernels to Run on Accelerators
NsNet2	Linear GRU, Matmul Sigmoid ReLU
ConvTasNet	Conv1D ConvTranspose1D Sigmoid PReLU GroupNorm Conv1D: point-wise, depth-wise, causal
Yolov6	Conv2D
Yolov7	Conv2D
TCNN	Linear Conv1D ReLU BatchNorm1D PReLU Conv1D: point-wise, depth-wise, causal
Other/Shared	STFT ISTFT Element-wise ops (sum, prod)
Transformers	Multi-head cross-attention, self-attention

Table 2.1. A summary of CONVOLVE partner use cases and additional applications, focusing on speech and audio processing and the important kernels.

2.2 Frameworks for NNs

In this section we discuss mainstream neural network frameworks and dedicated optimizations libraries.

2.2.1 PyTorch

PyTorch is a high-performance tensor library for deep learning on CPUs and GPUs, enabling efficient computation graph construction and execution. It is highly optimized for mainstream architectures such as Intel and AMD. At its core, PyTorch uses the ATen library to handle core CPU/GPU operations. For x86 CPUs, it integrates external, hand-optimized libraries such as MKL and MKL-DNN, which exploit hardware features for Eg., AVX512 for improved SIMD and memory efficiency. In contrast, RISC-V support in PyTorch remains limited. Many dependencies are unavailable or require extensive rework, and hardware-specific optimizations are largely absent. As a result, performance on RISC-V lags significantly behind that of more mature architectures.

2.2.2 ONNX

ONNX is an open-source format for artificial intelligence models, including deep learning and traditional machine learning. ONNX aims to provide a common language that any machine learning framework can use to describe its models. ONNX implements a Python runtime that can be used to evaluate ONNX models and to evaluate ONNX operations. This is intended to clarify the semantics of ONNX and help us to understand and debug ONNX tools and converters. It is not intended to be used for production, and performance is not a goal.

2.2.3 ONNX Runtime

ONNX Runtime (ORT) uses efficient graph optimizations on ONNX graphs, such as removing redundant nodes and computations, statically computing constants, and fusing multiple nodes into a single node. These optimizations run before partitioning the graphs into subgraphs, ensuring they apply to all Execution Providers (EP). Furthermore, ORT optimizes sub-graphs for different hardware configurations based on the assigned Execution Provider (EP), e.g., convolution and activation fusion on CPU EP. These optimizations are specific to the assigned EP. This approach maximizes performance across CPUs and ORT supported EPs. We aim to harness the insights gained from existing frameworks in our compilation pipeline. While many of these optimizations have been manually fine-tuned, we aim to bring these optimizations to our compiler and tailor them in an automatic manner.

2.3 Compilation Frameworks

This section provides a brief overview of the frameworks we extend and build our compilation flow on. The specific features relevant for this deliverable are highlighted.

2.3.1 LLVM

LLVM is a modular compiler infrastructure designed to support program analysis and transformation. At its core is the LLVM Intermediate Representation (IR), a low-level, typed, assembly-like language that enables language-independent analysis and optimization. LLVM is widely used for static analysis because it provides rich APIs and built-in analyses for control flow, data flow, memory access, and more. By operating on LLVM IR, developers can write powerful static analysis tools that detect bugs, verify program properties, or optimize code. Moreover, LLVM is a robust tool that supports a variety of hardware platforms and can be easily extended for new targets.

2.3.2 MLIR

MLIR (Multi-Level Intermediate Representation) provides strong support for neural networks and transformers through its flexible, dialect-based design. It natively integrates with ONNX, enabling optimized compilation and code generation for a wide range of classic neural network models. For transformers, MLIR supports key components like multi-head attention and feedforward layers, facilitating efficient optimization and lowering from popular frameworks such as PyTorch via projects like Torch-MLIR. Its extensible transformation pipelines allow fine-grained optimizations critical for large-scale models, including vision transformers and diffusion transformers. Overall, MLIR serves as a powerful, adaptable compiler infrastructure bridging high-level ML frameworks and diverse hardware targets, accelerating neural network and transformer deployment across platforms.

2.3.3 IREE: An ML Compiler and Runtime Framework

The IREE framework is a unified end-to-end compiler and runtime stack that enables flexible deployment of ML workloads to hardware targets of varied heterogeneity and scale. Its focus on ML model portability allows targeting effectively both resource-limited platforms and larger deployment targets. IREE follows a comprehensive approach to ML compilation by offering:

- support for a wide range of advanced ML formats (e.g., ONNX, TOSA, etc.) and their features (e.g., streaming, dynamic shapes, etc.),
- code generation of IRs that encapsulate both scheduling, which manages aspects of host and target device communication (e.g., device placement, data dependencies), and execution logic, which encodes computation tailored to the target accelerator.

By integrating with MLIR, IREE can leverage its rich abstraction capabilities across the compilation stack, enabling flexible specialization and retargeting at every level.

2.4 Static Analyses

Static Analysis is the process of examining code without executing it, typically to find bugs, enforce coding standards, or verify program properties. Static analysis can range from simple pattern matching to advanced techniques such as data flow analysis and symbolic reasoning,

making it a crucial tool in building correct, reliable, and secure software. Examples include: pointer analysis, use-def, SCEV (refer to section [3.2.4 Instruction reordering at LLVM IR for Instruction fusion](#)).
LLVM and MLIR support static analysis tools and passes that can gather program characteristics, such as instruction counts and memory access patterns, to inform or drive transformations.

2.4.1 Design Space Exploration

ZigZag is a DSE framework which explores matching a neural network layer (or full neural network using a layer-by-layer schedule) to a single processing element (PE) array-based accelerator. Given a hardware description and an NN workload, ZigZag outputs a recommended tiling scheme according to its cost model.

UMU's cost-model-driven tiling takes inspiration from KUL's DSE framework ZigZag.
In the following subsections, we highlight:

- Our efforts manually integrating ZigZag into the compiler (3.3.3)
- KUL's complementary work Praxis, which fully integrates ZigZag into an MLIR-based compiler (3.3.4)
- Myrtle, which provides the compiler with automatic tiling for the Snitch Cluster (3.3.5)

2.5 Roofline Model

The Roofline Model is a useful performance analysis tool that relates computational throughput to memory bandwidth. It plots arithmetic intensity (operations per byte moved) against achievable performance, revealing whether a model's layers are limited by memory access or compute capacity. By showing these bottlenecks, the Roofline Model helps guide optimizations—such as improving data locality or increasing parallelism—to enhance neural network efficiency and better utilize hardware resources.
Deep learning workloads, such as convolutions and matrix multiplications in NNs, often shift between memory-bound and compute-bound regimes depending on model size, batch sizes, and kernel optimizations. The Roofline Model helps identify if a given neural network layer or kernel is underperforming due to insufficient optimization.

2.6 Static transformations

Both MLIR and LLVM provide extensive support for static code transformations through their modular pass infrastructure. These transformations are applied at compile time, operating on the program's intermediate representation (LLVM IR) or on specific dialects (MLIR) to optimize, refactor, or instrument code before execution. MLIR and LLVM are designed for extensibility, providing robust infrastructure for adding custom transformations, primarily through their pass framework. Developers can implement their own transformation passes to analyze or modify the IR and dialects, at various stages of the compilation pipeline. We leverage MLIR's and LLVM's support for static transformations, which is robust, extensible, and central to their

design. In particular, we target the following optimizations and the afferent transformations. We detail here the main neural network optimizations we focus on in Convolve.

2.6.1 Tiling

Tiling is the process of breaking the data access pattern of a loop into smaller, regular-sized pieces (referred to as blocks or tiles) to improve spatial and/or temporal locality [15]. This method can also be used to achieve parallelism, where computation inside a loop is broken up into pieces and distributed to multiple cores [1].

2.6.2 Vectorization

Vectorization accelerates computation by performing operations on multiple data points simultaneously using SIMD (Single Instruction, Multiple Data) hardware. Auto-vectorization enables compilers to optimize code automatically for such parallel execution. Machine learning and neural network workloads are ideal for this, as they involve large, repetitive data operations well-suited for vector processing.

2.6.3 Static Instruction scheduling for fusion

Instruction reordering is a powerful compiler optimization that rearranges instructions to better exploit the underlying microarchitecture. It enhances performance by exposing memory-level parallelism (MLP), reducing cache miss latency, and increasing instruction-level parallelism (ILP). Reordering can also enable vectorization by aligning data operations and support instruction fusion (macro-op fusion), where adjacent instructions are combined into a single micro-op to reduce decode and execution overhead. By leveraging static analysis and program semantics, the compiler can perform these transformations without runtime speculation, offloading complexity from the CPU and improving both performance and hardware efficiency.

2.7 Quantization and Mixed-Precision Quantization

Quantization is a technique for compressing neural networks that leverages a mapping from full-precision data types (32- and 64-bit floating points, used to train the neural network) to smaller data types, usually 8- or 16-bit integers.Quantization leads to a significant storage reduction since the network's weights and activations are represented with a reduced number of bits. Furthermore, operations between integers are notoriously less expensive than those between floating points, regarding both computational effort and energy consumption. However, quantization negatively affects the network accuracy, as approximating on fewer bits can potentially compromise the neural network performance.

Nevertheless, different network weights and activations have diverse sensitivities to the perturbations introduced by quantization. As a result, uniform quantization is simple and fast to implement but not optimal. From this consideration, mixed-precision quantization (MPQ) has been designed. It allows quantization with different precision in the various layers of the network. This leads to an optimal trade-off between accuracy and hardware performance. MPQ

is complex to implement, as it requires a sensitivity analysis to define how the different layers react to different aggressivity of quantization without compromising the network performance.

2.8 Memory Allocation

In the CONVOLVE compiler, memory allocation is conducted in three stages, all of which are parts of the IREE backend: first, tensor operands and results are bufferized and lifetime annotated. Then, memory buffers whose size is known at compile-time receive offsets to minimize total memory usage. Last but not least, buffers whose size is resolved at runtime are also annotated with offsets—but these are computed online during inference.

In general, offset assignment to lifetime-annotated buffers is NP-complete. ICCS has developed `idealloc` [12], an elastic allocator, achieving state-of-the-art performance. The following project deliverables describe its aspects:

- D5.6 "Final memory management and allocation for ULP accelerators" (M33) → background, design, implementation and evaluation
- D5.4 "Integrated tool flow" (M33) → integration with IREE

3 Static Analysis Tools and Optimizations

We have first analyzed the performance of the Convolve use cases when optimized through state of the art frameworks for neural network optimizations. We identified that while these frameworks have been heavily fine-tuned for standard off-the-shelf architectures such as x86, they do not keep up with other architectures, such as RISC-V, not to mention emerging accelerators. Based on this observation, we assembled an end-to-end compiler and designed a series of optimizations for neural networks for RISC-V platforms, focusing on key optimizations such as data and code layout transformations and other compiler extensions, which can then enable automatic vectorization, efficient tiling and kernel mapping, instruction reordering for fusion, etc. All such transformations require powerful static analyses to ensure the correctness (validity) of the transformation and that it is beneficial.

This section describes our contributions, as highlighted in the executive summary (ES):

- Analysis of optimizations offered by State-of-the-Art frameworks
- OML and OML-vect proposal to generate executables for RISC-V efficient vectorize
  Specifically:
  - Reduction pass
  - Data layout optimization pass
- Reorganizing code to alleviate performance bottlenecks and improve real-time guarantees.
  Specifically:
  - Scheduling instruction to facilitate fusion, which maps efficiently to the core pipeline, hence improving performance, energy and resource utilization (Sec 3.2.4) .
  - Validating the correctness of compile-time transformations, identifying hot functions (Sec 3.2.5).
- Compile time support to analyze and reorder instructions in general purpose languages, which is significantly more challenging than optimizing compiler-friendly neural network kernels (Sec 3.5).
- Cost-Model Driven Tiling
  - Manual integration of the ZigZag DSE tool into the compiler (Sec 3.3.3).
  - Identifying Snitch compatible kernels within a larger Neural Network and offloading them appropriately to the snitch cluster. (Sec 3.3.4.1 and 3.3.5).
  - Automatic tiling and loop interchange for execution on the Snitch Cluster, optimized for L1 cache and register reuse. (Sec 3.3.5.2).
  - Sensitivity Analysis to reveal trends in optimal tile size (Sec 3.3.5.1).

3.1 Analysis of optimizations offered by State-of-the-Art frameworks when applied to the Convolve use cases

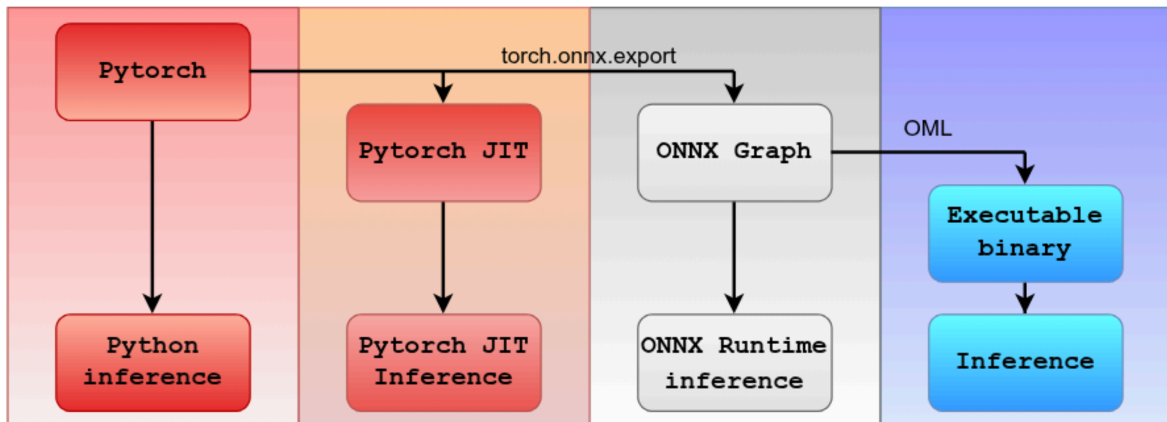


Figure 3.1: Pipeline illustration to compile/interpret neural network workloads using PyTorch, ORT and OML

Since 2019, PyTorch has seen widespread adoption due to its dynamic computation graph, ease of debugging, parallelism, and a rich ecosystem of hand-optimized libraries. In contrast, ONNX Runtime (ORT) focuses on high-performance inference, prioritizing speed and cross-platform efficiency over development flexibility. We decided to evaluate the performance and analyze the optimizations of PyTorch and ORT to understand their strengths and limitations with a focus on neural network workloads, as shown in Figure 3.1

We have analysed the way both PyTorch and ORT leverage external libraries such as MKL-DNN and XNNPACK to exploit hardware-specific optimizations. MKL-DNN functions such as mkldnn_relu, mkldnn_max_pool2d, and mkldnn_binary_fusion utilize vector extensions such as AVX512 to improve performance through parallelism and efficient memory use. PyTorch's fusion_unary_attr_map enhances this by fusing unary operations (e.g., ReLU) with attribute mapping, reducing data movement. Similarly, mkldnn_binary_fusion optimizes convolutions by fusing them with binary operations, boosting performance for complex neural networks on x86 CPUs. However, porting PyTorch to RISC-V faces challenges such as missing dependencies (e.g., cpuinfo) and the need to rewrite architecture-specific optimizations, leading to a significant performance gap compared to x86. ORT supports RISC-V with selective hand-optimized kernels for operations such as GEMM and convolution, but coverage is limited. For instance, in the nsnet2 Jabra GRU use case, only GEMM is optimized, leaving tanh untouched.

In our analysis, we discovered that while both frameworks are optimized for layers such as convolutions and matrix multiplications essential to models for Eg., AlexNet and MobileNet, fewer optimizations exist for RNNs, LSTMs, and Transformers. Extending support for these components on RISC-V requires substantial engineering effort, widening the performance gap with established architectures. The Convolve accelerators are built on top of the RISC-V ISA, thus we focus on providing an end-to-end compilation chain for RISC-V and dedicated optimizations, as detailed below.

3.2 Auto-vectorization

Auto-vectorization is an optimization technique that automatically converts scalar operations within loops into vector operations, enabling parallel execution. This process is key to improving performance, especially for loop-heavy tasks such as those found in neural networks.

In MLIR, auto-vectorization is facilitated through passes such as the affine super-vectorizer, accompanied by helper passes such as "the reduction pass". These passes analyze loop nests, identify opportunities for vectorization, and transform them into efficient vector operations, maximizing hardware performance, and reducing execution time.

3.2.1 Study of the factors that prohibit vectorization at LLVM IR

We analyzed the LLVM vectorization pass reports for the nsNet2 baseline model and found that 37 loops were considered for vectorization and only 11 were vectorized. Among the top reasons that prevented vectorization were: (i) unsafe floating-point operations (even with ffastmath flag), (ii) scalar cost being optimal, and (iii) unvectorizable types such as function calls in the loop body. We show that through specific optimizations such as loop fusion, partial loop unrolling, and effective vectorization at both the MLIR and LLVM IR levels, we are able to further increase neural networks' performance.

3.2.2 Study of the factors that prohibit vectorization at MLIR

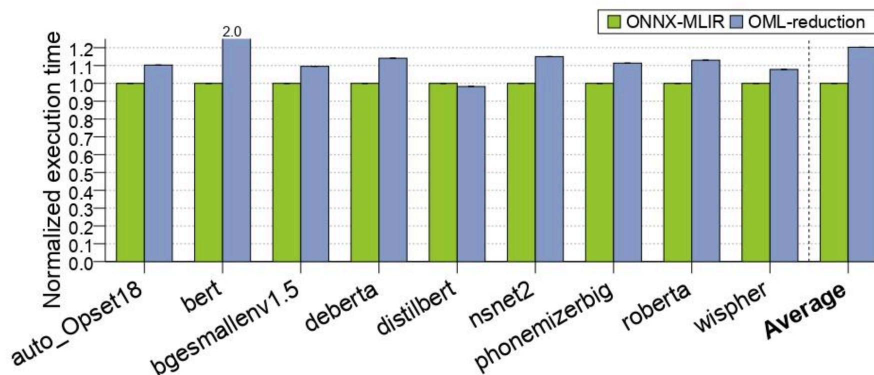


Figure 3.2.2: Performance degradation on on Intel Xeon (x86) due to forceful vectorization when data access is non-consecutive

Kernel	Dialect	Reduction op.	Identified
MatMul (float)	math	fma	No
MatMul (float)	arith	addf	Yes
MatMul (int)	arith	addi	Yes
SoftMax	arith	maxnumf	No

Table 3.2.2: Ability of MLIR to identify common reduction operations in Transformers

The MLIR framework "match-reduction-test" provides a reduction pass designed to identify simple reduction patterns, such as addition ('addf') operations from the 'arith' dialect. However, it fails to recognize more complex reduction patterns, such as reduction operations from the 'math' dialect, such as fused multiply-add (FMA), maximum, or minimum operations. In certain scenarios, it may even fail to identify 'addf' operations. Table 3.2.2 summarizes the reduction operations commonly found in Transformer models, indicating whether the MLIR reduction pass is capable of identifying or fails to recognize them. In addition, the reduction pass operates at the function level and produces results that cannot be directly passed to the affine super vectorizer. Vectorization is beneficial when the memory accesses are contiguous, while non-contiguous memory accesses can lead to performance degradation. Figure 1 illustrates a basic pseudocode for MatMul, clearly showing that matrix A has contiguous memory accesses, in contrast to matrix B. MLIR's 'vectorizeLoops does not vectorize MatMul due to its data access patterns. Forcing vectorization against the indication given by the cost model leads to performance degradation as shown in figure 3.2.2.

3.2.3 State-of-the-art: ONNX-MLIR with *manually* vectorized libraries

Input: Matrix A of size $m \times k$, Matrix B of size $k \times n$
Output: Matrix C of size $m \times n$ where $C = AB$
VL = Vector Length

for $i = 0$ to m:
  for $j = 0$ to n with step size VL:
    $C[i][j : j + \text{VL} - 1] = 0$
    for $l = 0$ to k:
      $A_brd = vector_\text{broadcast}(A[i][l], \text{VL})$
      $C[i][j : j + \text{VL} - 1]$ **+=** $A_brd \circ B[l][j : j + \text{VL} - 1]$

Figure 3.2.3 Vectorized ONNX-MLIR MatMul

Figure 3.2.3 illustrates the current approach in ONNX-MLIR for vectorizing MatMul operations. Assuming a matrix multiplication C=AB, for each element of matrix A, the corresponding row of matrix B is selected. Specifically, for element A[i][k], the k-th row of matrix B is chosen, and the value of A[i][k] is multiplied by each element in the k-th row of B. This multiplication is carried out using vectorized operations to enhance performance. Unlike basic MatMul where B is accessed column-wise. The decision to vectorize using libraries that have been manually optimized stems from MLIR's lack of a reduction pass to identify a wide range of reduction ops and an affine vectorizer unable to vectorize MatMul due to data access patterns. Although the ONNX-MLIR's manual optimizations ensure consecutive memory accesses for both the input matrix A and B, it significantly increases the number of loads, stores, and required vector broadcast operations. For a matrix multiplication with a A matrix of size 1x1x251x257 multiplied by B of size 257x400 (a common case found in the nsnet2 usecase), this approach results in 1,177,056 vector loads and 589,056 vector stores (in affine dialect). Consequently, increasing the number of loads and stores can lead to memory bottlenecks, weakening performance due to

excessive data transfers. Optimizing memory accesses is crucial to improve computational efficiency.

3.2.4 Our proposal: OML-vect

We propose a technique that addresses limitations in the MLIR framework in order to enable and enhance auto-vectorization. The pipeline begins by exporting the pre-trained neural network to ONNX graphs. The ONNX graph is then traversed, and its data layouts are modified to optimize the code for auto-vectorization. Subsequently, the graph is lowered to the ONNX dialect, followed by the krnl dialect, and finally to the affine dialect using ONNX-MLIR. At the affine dialect level, reduction patterns are identified and the "Affine loop-vectorizer" is invoked. The vectorized code is then progressively lowered to SCF, CF, and LLVM dialects, ultimately generating LLVM IR, assembly (asm), and eventually standalone binaries are generated.

3.2.5 Contribution: Data layout optimization for vectorization

We traverse the ONNX graph bottom-up to identify operations that hinder auto-vectorization due to inefficient data access patterns. A common example is matrix multiplication (MatMul, A × B). To improve vectorization, we transpose the second matrix (B).

In Transformers, MatMuls fall into two categories based on the second matrix's role:

Weighted MatMuls:
If the second matrix (B) is a constant weight known at compile time (Figure 5a, MatMul (1)), we transpose B during compilation and annotate the MatMul node with `transB = 1` (Figure 5b, MatMul (1)). Since these weights are used only within this kernel, the original matrix is discarded, reducing memory usage.

Generic MatMuls:
If both A and B are dynamic inputs (Figure 5a, MatMul (2)), we optimize differently. Typically, a transpose precedes B. We traverse the graph to locate this transpose, stop at non-scalar ops, and adjust its permutation instead of inserting a new transpose. This follows the "Composition of Two Transpose Operations" principle [7], simplifying the graph and avoiding extra overhead. We ensure the transpose is exclusive to this MatMul to prevent side effects. Updated metadata marks the MatMul as using a transposed B (Figure 5b, MatMul (2)).

Altering B's layout requires adapting the MatMul algorithm as shown in figure 3.2.5A .

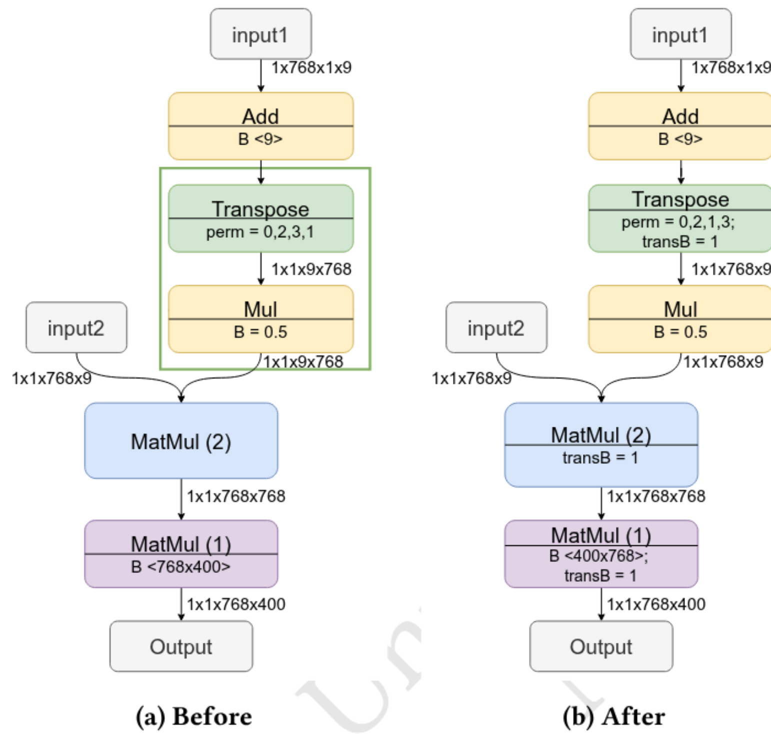


Figure 3.2.5A: ONNX graph before and after data layout optimization

Input: Matrix A of size $m \times k$, Matrix B of size $k \times n$
and $B_Transposed = Tranpose(B)$
Output: Matrix C of size $m \times n$ where $C = AB$

if $transB = 1$:
for $i = 1$ to m:
  for $j = 1$ to n:
    $C[i][j] = 0$
    for $l = 1$ to k:
      $C[i][j] += A[i][l] \cdot B_Transposed[j][l]$
else:
for $i = 1$ to m:
  for $j = 1$ to n:
    $C[i][j] = 0$
    for $l = 1$ to k:
      $C[i][j] += A[i][l] \cdot B[l][j]$

Figure 3.2.5B OML-vect Matrix Multiplication

While lowering the ONNX graph to affine dialect, the transpose metadata flag dictates how the matrix multiplication is performed. If set, OML-vect swaps the indices of the B matrix during MatMul as shown in Figure 3.2.5B; if the flag is not set, the standard MatMul algorithm is used.

3.2.6 Limits of the reduction detection in MLIR and our solution

Reductions are essential operations used to combine values across loop iterations, such as summing elements in an array or finding the maximum value in a dataset. Efficiently handling reductions is crucial for achieving optimal performance, particularly in loop-intensive programs. In MLIR, a reduction pass helps identify and optimize these operations within loop nests.
Affine dialect provides auto-vectorization functions, which we utilize to enhance performance. The process begins by identifying the innermost loop, detecting potential reductions, and invoking the auto-vectorizer. The reduction pass starts by identifying the innermost for-loop in the affine representation, focusing on loops with iter_args, which are values updated during each iteration and returned by the loop. These updates use combiner operations such as add, multiply, or max, which correspond to known reduction patterns. If a complex operation consumes iter_args but is not directly recognized as a reduction, we check if it can be broken down into simpler operations that are. For example, a fused multiply-add can be split into a multiply and an add, allowing us to detect the reduction. We also check input and output shapes to confirm reductions, such as when a tensor is reduced to a scalar. Using the parent operation, combiner type, and iter_args, we build a reduction map that describes the reduction behavior. Finally, the super vectorizer optimizes the detected reductions, and the resulting code is lowered to LLVM.

3.3 Automatic Tiling

We present automatic tiling techniques from two complementary MLIR flows:
- Myrtle provides automatic tiling for the Snitch Cluster target (UMU)
- Praxis provides automatic tiling to a selection of KUL's systolic array-based accelerators (KUL) (see subsection 3.3.4)

In Myrtle, we see examples of
- Identifying coarse-grained opportunities to offload computations to fixed-function accelerators via the ConfigureForTiles pass of Quidditch (section Myrtle 3.3.5)
- Test-driven generation of optimization opportunities in Sensitivity Analysis (section 3.3.5.3)
- Remapping data layouts to minimize transfers through maximizing L1 cache usage and favoring snitch streaming registers when selecting a tiling scheme (section 3.3.5.2)

3.3.1 State-of-the-art in tiling

Compiler infrastructure tools tend to punt tile size selection to users, or opt for a parameterized tiling pass that handles tiling for parallelism only.

MLIR provides an automatic tiling pass at the affine dialect level, but leaves tile sizes as user input [8]. It used to also provide an automatic tiling pass at the linalg dialect level, but due to the infeasibility of a one-size-fits-all solution, developers decided to remove the tiling pass so as not to mislead users about its applicability [18]. Today, MLIR provides library functions to users that perform the tiling transformation on a linalg operation, provided tile sizes are also

supplied as input [29]. Users of MLIR are expected to build their own tiling pass using these functions [18].

IREE provides a tiling pass for CPUs, but requires the user to annotate the input IR with tile sizes ahead of time [9]. IREE's GPU pipeline provides automatic tiling for parallel distribution. Tile sizes are parameterized on warp size, which is set to 32 by default [10]. IREE provides an autotuning framework called SHARK Tuner users can experiment with to find better compilation settings for their dispatches, for example, finding optimal tile or warp sizes for each dispatch in their input program [25]. However, this process requires offline tuning: users pass these settings in a spec file along with their input program to the iree compiler.

3.3.2 Tiling in Quidditch

3.3.2.1 Quidditch Dispatches

The IREE front end converts Pytorch Neural Network layers (figure A.a,b) into a series of linalg dialect operations (figure A.c,d), then groups these operations into self-contained functions called dispatches (figure A.e,f). Every dispatch contains what IREE calls a "root operation". The root operation is identified using heuristics, but is usually the operation in the group that contains a reduction [26].

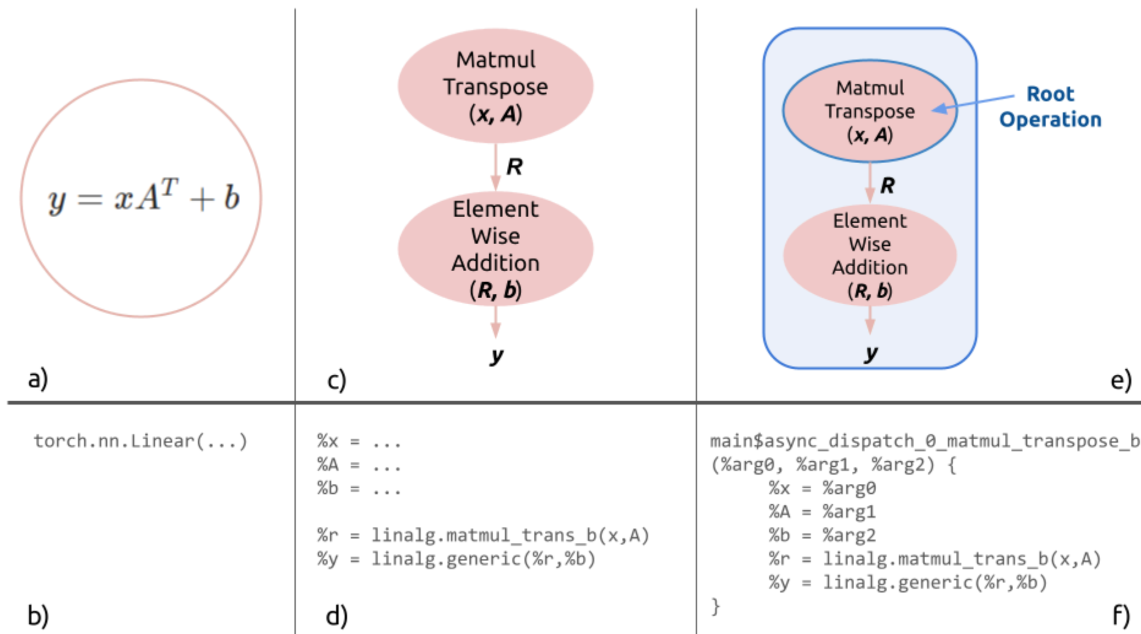


Figure A.Visual and Pseudocode representations of a NN Linear Layer at 3 levels of abstraction (left to right): Pytorch, Linalg MLIR, IREE Dispatch

The Quidditch backend compiles these dispatches and deploys them on the Snitch cluster.

3.3.2.2 Tiling Dispatches

Quidditch supports two levels of tiling, the first for data locality (denoted as "L1" level tiling), and the second for parallelism (denoted as "Thread level tiling"). Thread level tiling is simple; the largest parallel dimension of the operation gets tiled, with tile size determined by the quotient of the largest parallel dimension and the number of cores in the Snitch cluster. L1 level tiling, which emphasizes data reuse, is more intricate and employs the Myrtle cost model to determine tile sizes.

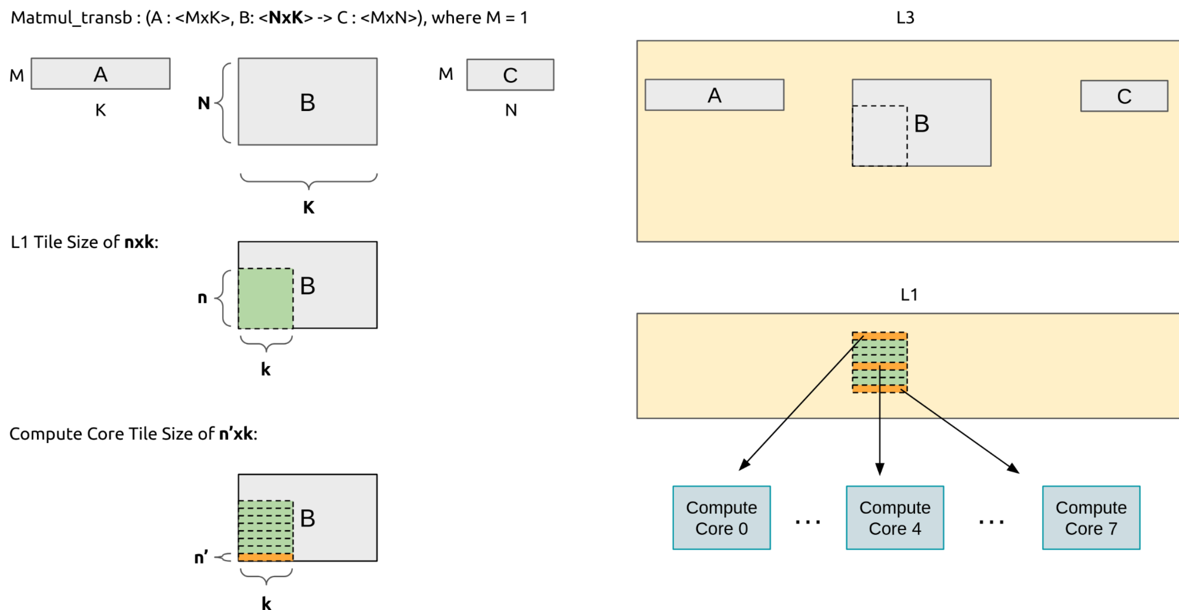


Figure B. Quidditch supports two levels of tiling. Given a tiled matmul transpose operation with input size MxNxK, M=1, L1 Level tiles of the B opearnd have size nxk, and Thread Level tiles have size n'xk'. Note that the N dimension gets tiled twice, once at the L1 level then again at the Thread level. Tile for the A and C operands omitted for clarity.

Quidditch tiles on a per-dispatch basis, meaning the tiles for every operation within a dispatch must fit in L1 at the same time. Root operations are tiled at the L1 level, then once more at the Thread level. Non-root operations are tiled only at the Thread level.

3.3.2.3 Double Buffering L1 Tiles

Quidditch employs double buffering to speed up kernel execution. In figure B, we observe a B-operand tile with dimensions nxk copied from L3 to L1, and processed by the 8 Snitch compute cores. With double buffering, instead of waiting for the cluster to complete computation on the first tile, the DMA core immediately starts copying the next tile of B into a second buffer inside L1. When the compute cores finish processing the first tile, the cluster of compute cores switch to processing the next tile in this second buffer. Alternating between these two buffers reduces the overall time compute cores spend waiting on DMA transfers. As a trade-off, space for two B-operand tiles (two "buffers") must be allocated in L1 during kernel computation.

3.3.3 Proof of Concept: Manual ZigZag-Compiler Integration (UMU)

UMU created manual examples of ZigZag prescribed tile and loop interchange transformations (tiling schemes) at the MLIR level. By starting with manual MLIR-based static analysis and transformations, we hoped to easily target both the snitch cluster and snax accelerators. However, due to the rapid evolution of the snax accelerators, in-progress status of snax-mlir, and steep learning curve for a combination of tools, we shifted our efforts to solely target the snitch cluster, which provided a more stable compilation flow at the time. Our work culminated in an NsNet kernel tiled with a ZigZag prescribed tiling scheme (tiling scheme itself obtained offline) running on the Snitch Cluster using EDI/CAM's Quidditch flow.

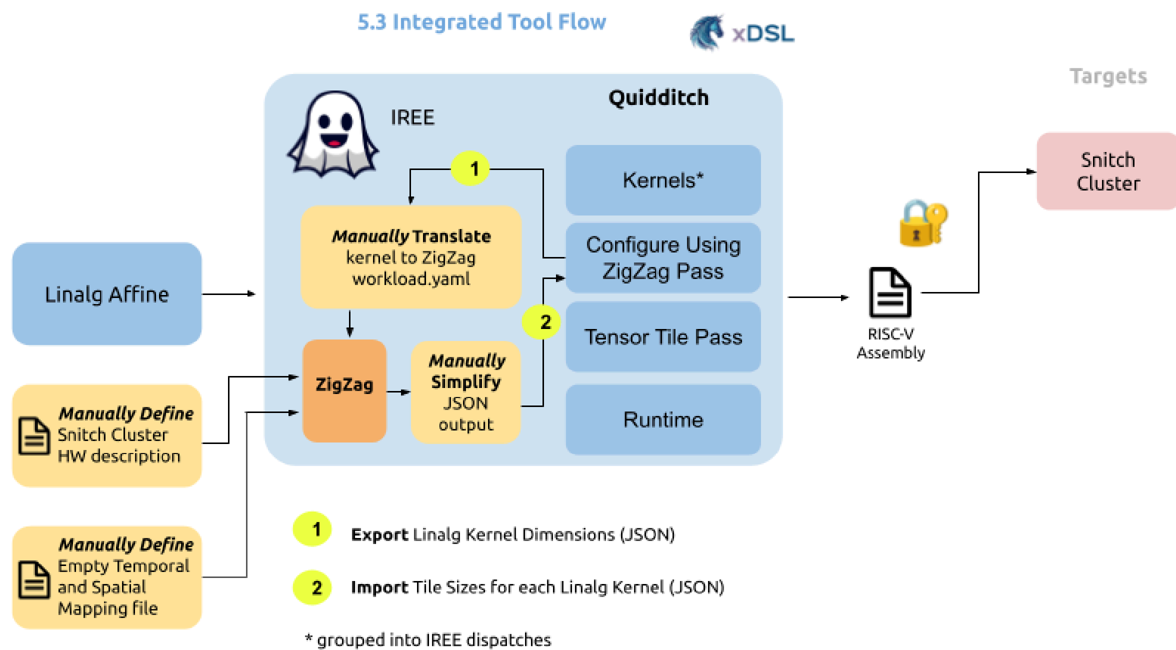


Figure A. Manually integrating ZigZag into the compiler.
For more details on these efforts, refer to deliverable 5.4 Integrated Tools, section 3.5.1.

After an in-depth sync with KUL in the fall, we learned that some of the work linking ZigZag-prescribed tiling schemes with MLIR had been taken up by the KUL team as part of a different project (see section 3.3.4 and subsection 3.3.4.7 Praxis), and that ZigZag's cost model (designed for accelerators with dense arrays of processing elements) was unsuitable for targeting snitch compute cores. For these two reasons, **UMU pivoted to developing Myrtle, a ZigZag-inspired tiling cost model for the Snitch Cluster (see section 3.3.5)**. This pivot successfully avoids work duplication, and concentrates UMU's efforts on providing necessary, relevant, and new static analysis to the CONVOLVE project.

3.3.4 ZigZag-Compiler Integration

For an HW-accelerator targeting compiler toolchain, it is important to optimize the spatial and temporal scheduling of the workload. These scheduling aspects can have a humongous effect on the required amount of data movements, and as such on the execution efficiency. This is possible with the ZigZag and Stream tools, developed in WP6. It however requires a tight linking between the MLIR compile toolchain of WP5, and the exploration tools of WP6.

ZigZag is a design space exploration tool, which enables automatic generation of optimal tile configurations and temporal mapping strategies. These solutions address multiple key challenges simultaneously: adapting memory access to limited scratchpad capacity, minimizing data movement between the accelerator and off-chip memory, and automating tile-scheduling decisions across diverse DNN layers. This should now be exploitable by the compiler, all with minimal programmer effort.

3.3.4.1 Compiler Targeting of Stream-Oriented Accelerators

The compiler toolchain targets a stream-programmable accelerator architecture in which computational datapaths are tightly connected to a unified shared memory via configurable affine data streamers. These streamers execute affine address generation logic and decouple compute from data layout, enabling a highly flexible flow of operand data regardless of the underlying dataflow strategy.

The architecture supports reconfigurable loop nest execution patterns, specified via programmable affine access streams. Computation on the accelerator is performed without fixed assumptions on spatial or temporal mappings. As such, the same hardware may be deployed across a wide variety of matrix configurations, convolution kernels, and memory layouts using proper stream configuration.

To operate this architecture efficiently, the compiler must generate mappings from high-level tensor operations to loop-scheduled, tiled affine streams. This is where the use of ZigZag proves critical (see also WP6).

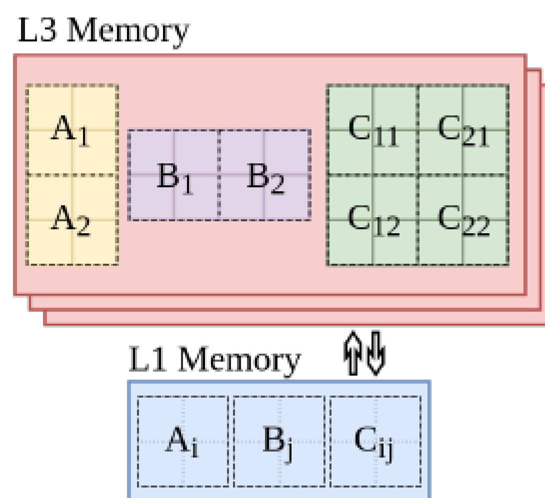


Fig 3.3.4.1: The need for tiling to adjust the matrix tiles to the buffer sizes at different levels of the memory hierarchy.

3.3.4.2 Leveraging ZigZag for Design-Space Exploration

ZigZag is incorporated into an earlier stage of the compilation flow, where it performs joint exploration of scheduling and memory usage across a tiling search space. Given a compute pattern such as a matrix multiplication or convolution and the physical constraints of the architecture, ZigZag's internal modeling engine estimates the performance implications of different tiling configurations.

The tool evaluates variations in tile size, loop ordering, and duration of memory residency, taking into account the structure of a memory hierarchy typically organized into slower and larger L3 memory and smaller L1 scratchpads. ZigZag's LoMA engine (Loop over Memory Allocation) selects configurations that maximize on-chip computation reuse while ensuring that no tile allocation exceeds the scratchpad boundary.

The outcome of this design space exploration is a spatial+temporal mapping: a multi-level, dimension-annotated loop nesting structure that reflects the best reuse behavior and compute–memory interaction pattern for the operation.

3.3.4.3 Automated Tiling in Compiler Middleware

To embed the result of this optimization inside the compiler, an intermediate representation framework based on MLIR is used. MLIR allows custom transformation passes to be inserted across levels of abstraction, and is extended here to manage progressive lowering of tensor operations such as matmuls into backend-specific loop nests and memory operations.

A key MLIR component is the Transform dialect, which provides a high-level construct, `tile_using_for`, for defining tile shapes. However, this dialect expects a simplified tiling form: each of the M, K, and N dimensions of a matmul may be tiled once per instruction. Meanwhile, the temporal mappings produced by ZigZag often consist of deeply nested loops, possibly tiling the same dimension multiple times in varying granularity.

To resolve this mismatch, the compiler first merges consecutive loops that target the same logical dimension, reducing redundancy while retaining correctness. It then partitions loop nests into groups, ensuring that no group tiles the same dimension more than once. These groups are lowered recursively in stages: each group results in a specialized `tile_using_for` instruction applied progressively, starting from the outermost level.

This layered transformation preserves the optimality of the tiling strategy while maintaining compatibility with the MLIR-based framework.

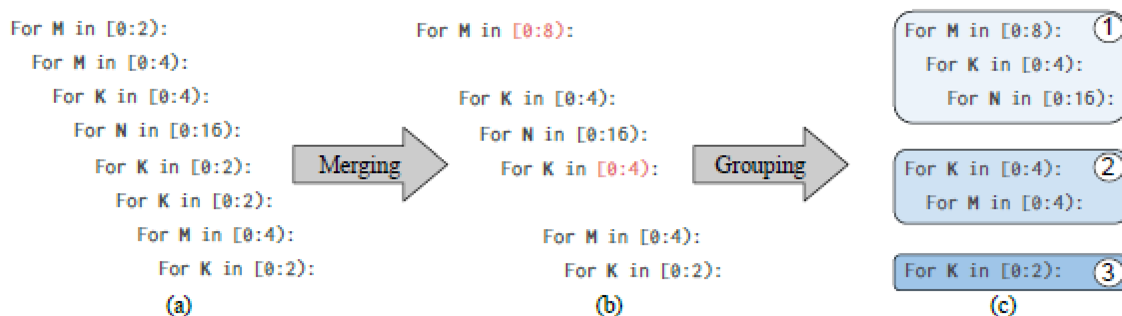


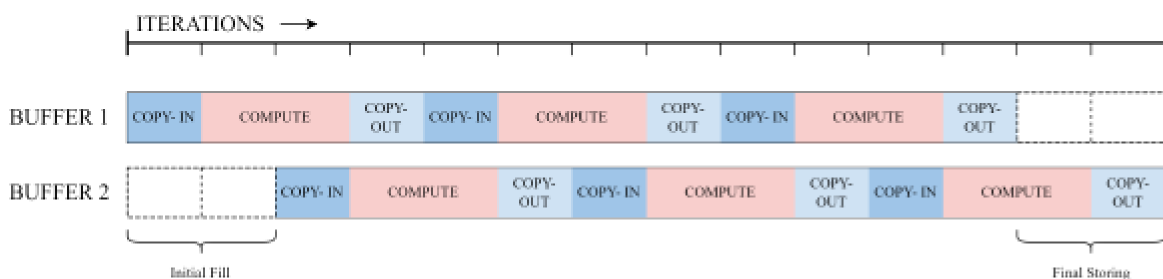
Figure 3.3.4.3: Adapting the found temporal mapping to a Transform-dialect legible state. (a) ZigZag-output, a list of for-loops. (b) Merging of consecutive loops along the same dimension. (c) Grouping of loops while avoiding the same dimension to occur twice in a group.

3.3.4.4 Managing Memory Reuse and Allocation in Tiled Execution

In tiled execution workflows, especially under strict L1 memory constraints, avoiding unnecessary memory reallocation is paramount. During default MLIR lowering, matrix operations allocate buffer space for each tile silently, often inside loop nests. This leads to exponential growth in memory usage—unsustainable for large DNNs.

The compiler introduces a buffer hoisting pass that analyzes tile buffer allocations within loops. When tile sizes are independent of the loop index, this pass relocates the allocation outside the loop, resulting in a single reusable buffer. In cases where tile size varies, typically due to boundary tiles not dividing the matrix evenly, the pass calculates the maximum tile size and pads smaller tiles with zeros to conform to that buffer configuration.

This memory reuse ensures that a fixed-size buffer is reused iteratively across tiles, with a consistent memory footprint.



Doubling buffering in time. Buffer 1 performs computations on loaded data while buffer 2 performs memory interactions. After an iteration is ready, the inverse situation start

3.3.4.5 Double Buffering to Overlap Memory and Compute

Once a fixed tiling and reuse strategy is in place, the opportunity arises to further enhance performance through double buffering. In tiled computations, there exists a clear sequencing of operations: loading data for tiles, computing results, and writing them back. If performed serially, these stages would block each other, stalling both computation and memory access.

Double buffering enables interleaving of memory and compute stages by using two alternating memory buffers. While the compute engine processes one tile from buffer A, the memory system loads the next tile into buffer B—and vice versa. This overlap minimizes idle time and maximizes throughput.

In the context of the compiler toolchain, a double buffering pass restructures the transformed loop nest such that every other iteration uses an alternate buffer. In the prologue and epilogue of the loop body, one-time loading and storing logic is inserted to handle the initial and final stages respectively. This transformation is valid only if buffer accesses and dependencies are isolated between tiles—guaranteed through earlier scheduling passes informed by ZigZag.

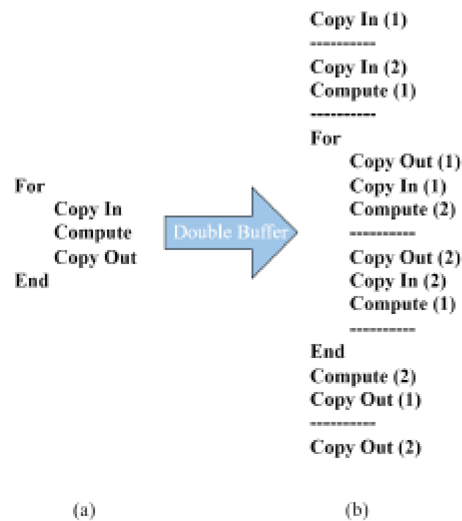


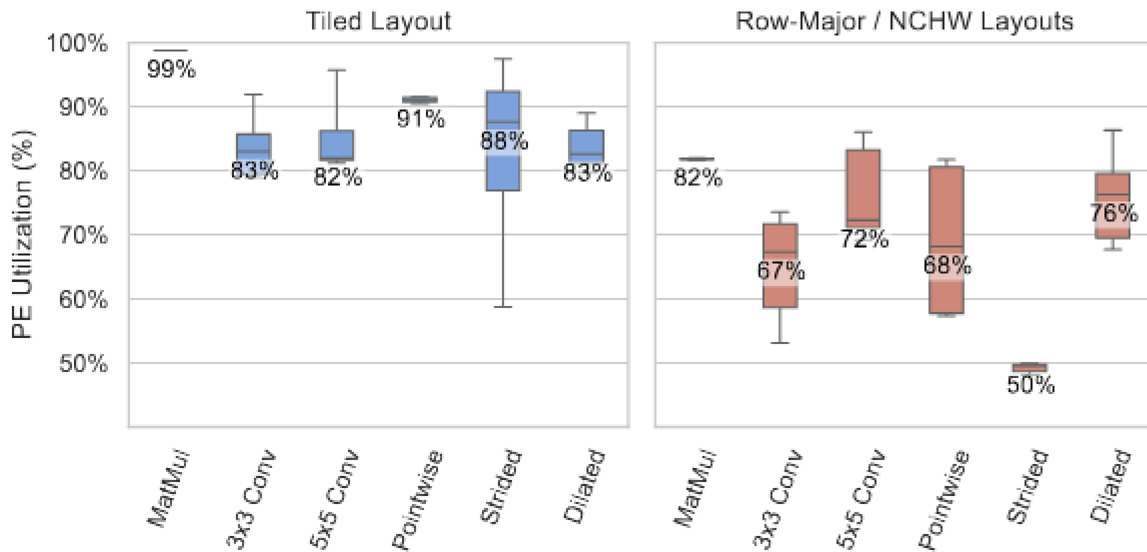
Figure 3.3.4.5: (a) General structure of a tiled linalg operation in pseudo-code. (b) Adapted structure of the tiled operation with double buffering applied, including the initial filling and final storing.

3.3.4.6 Evaluation and Deployment Benefits

Deployments of the full toolchain, combining programmable accelerators with compiler-controlled spatial/temporal mapping and ZigZag-optimized tile scheduling, have resulted in strong improvements in throughput and area efficiency for real-world neural models.

As shown in Figure XX, benchmarks on various compute kernels typical for modern neural networks show high processing element (PE) utilization—a measure for the throughput and hardware efficiency. The bars in the figure show the performance achieved across a wide range of kernels of different matrix dimensions, which are all tiled automatically. For each workload dimension, Zigzag explores all possible schedules as well as memory layouts, from which the compiler can then automatically deploy the most optimal one. The reported numbers are all utilization numbers measured on the actual hardware, after automated compiler mapping and deployment.

As can be seen, the deployment consistently gets a very high utilization of 80% or more. Moreover, the ZigZag-tiled layouts consistently the standard row major layouts used in industry. This results in roughly a 2× higher throughput compared to prior flexible accelerator generators using systolic arrays. This is attributed directly to the tight integration of memory-aware tile planning and stream configuration, largely driven by ZigZag’s design space exploration capabilities.



3.3.4.7 Outlook towards fully integrated tool

The integration of a design space exploration engine such as ZigZag into an MLIR-based compiler toolchain for stream-programmable accelerators enables a practical and fully automated path from high-level neural operations to low-level, execution on an existing hardware platform. Another aspect of the CONVOLVE project is, however, the automated generation of hardware with SNAX and OpenGeMM (see WP2 & WP6). The next step is hence to connect this ZigZag-driven compiler flow with the hardware generation flows. Here, Zigzag can not only explore the best schedule, but also the best hardware configuration. Next, the hardware is automatically generated, and the aligned code automatically compiled. Finally, the compiler code can run on the optimally generated hardware. This integration of 3 different toolsflows, is what is targeted by the Praxis framework, and reported on in Deliverable D6.4.

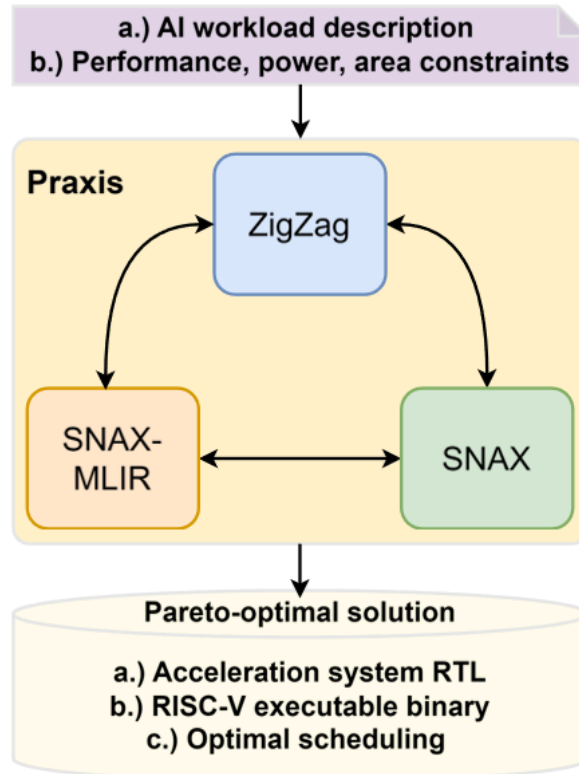


Figure: The Praxis framework

3.3.5 Myrtle: Automatic Tiling for the Snitch Cluster

The Myrtle Cost Model, paired with a modular tiling pass in Quidditch, automatically tiles NN linear layers to run efficiently on the Snitch Cluster.

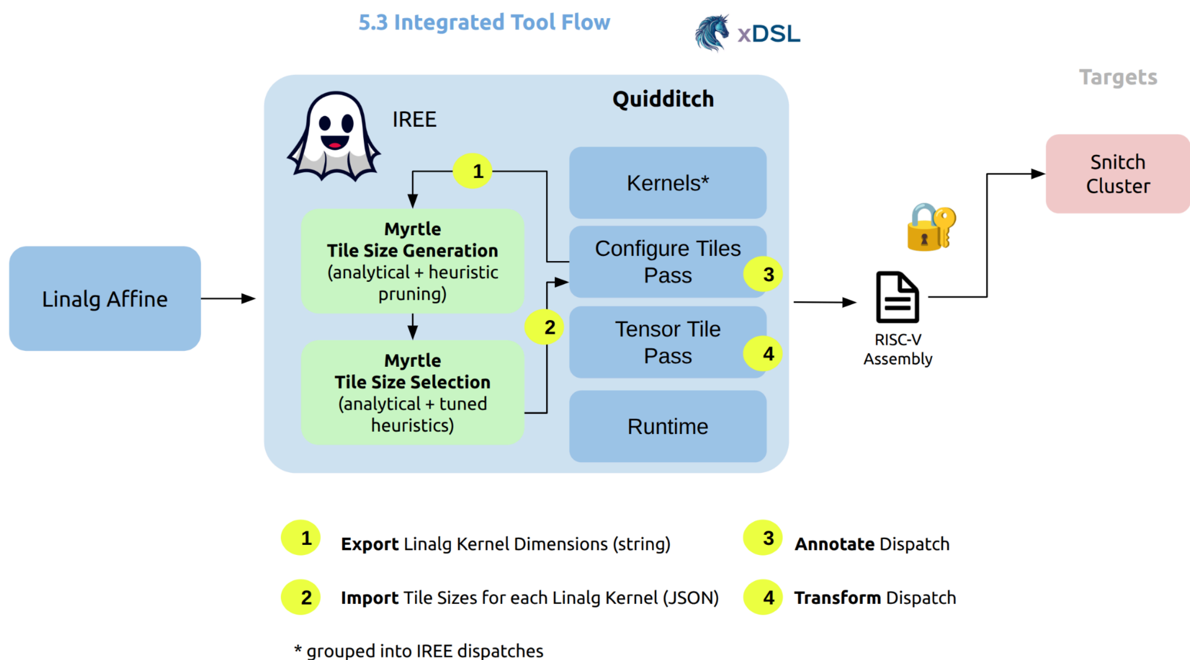


Figure A. Myrtle Cost Model + Automatic Tiling Passes. For more details about integration of Myrtle into Quidditch, refer to deliverable 5.4 Integrated Tools, section 2.5.1.3.

The ConfigureForTiles Pass identifies NN layers compatible for lowering with Snitch. When it finds a snitch compatible kernel, it queries Myrtle for the optimal tiling scheme (figure A, step 1), and finally marks them for offloading and tiling on the Snitch Cluster (figure A, step 2). After each dispatch has been correctly annotated with its root operation's respective tile sizes (figure A, step 3), the TensorTile pass performs the tiling transformation according to the tile sizes assigned (figure A, step 4).

3.3.5.1 Tile Size Generation

Myrtle prunes its search space by

1) Only considering thread level tile row dimensions that are a multiple of xDSL's snitch lowering's unroll-and-jam factor (between 1 and 7 inclusive).
2) Only considering L1 level tile row dimensions that are a multiple of 8 (so they can be evenly divided between the 8 snitch compute cores, a requirement of the Quidditch backend; validity check #1)
3) Only considering L1 level tiles which divide evenly into the input dimensions (with the exception of inputs with prime dimensions)
4) Only considering column dimensions up to half the input's column dimension, to allow for efficient double buffering.
5) Only consider tile sizes that fit in Snitch's L1 scratchpad (validity check #2)

3.3.5.2 Tile Size Selection

Myrtle selects the optimal or near-optimal tile size in three steps

1. Filtering by minimizing the number of microkernel runs (minimizes the overhead of streaming register configuration, which occurs once before each microkernel run)
2. Filtering by maximizing L1 usage
3. Filtering by minimizing regular loads from scratchpad to register (favors streaming register loads)

3.3.5.3 Sensitivity Analysis

To develop Myrtle, we first started by performing sensitivity analysis, comparing actual runs of neural network kernels with varied input and tile sizes. We then worked backwards from these insights, extracting the set of most significant contributing factors quantifiable at compile time.

Timing Entire Quidditch Dispatches (holistic runs)

We ran and timed NsNet2 from start to finish using the cycle accurate simulator verilator. During each start to finish "holistic" run, we also chose one Quidditch dispatch in which to vary the tile size, and timed its execution within the larger start to finish runtime.
Within the NsNet2 use case, five Quidditch dispatches can be lowered using xDSL's snitch backend, dispatches 0, 1, 7, 8, and 9:

Quidditch Dispatch	Kernels Inside	Input Sizes	Output Size
Dispatch 0	Vector-matrix transpose Element-wise addition	1x161, 400x161, 1x400	1x400
Dispatch 1	Vector-matrix transpose Element-wise addition	1x400, 1200x400, 1x1200	1x1200
Dispatch 7	Vector-matrix transpose Element-wise addition	1x400, 600x400, 1x600	1x600
Dispatch 8	Vector-matrix transpose Element-wise addition	1x600, 600x600, 1x600	1x600
Dispatch 9	Vector-matrix transpose Element-wise addition	1x600, 161x600, 1x161	1x161

Most Significant Factors, after Sensitivity Analysis:

- No padding in either dimension (see figure B)
- L1 usage during execution of dispatch (see figure C)
- Unroll and Jam Factor of xDSL microkernel lowering
- Double buffering
- Number of microkernel runs
- Number of regular riscv register loads from L1 (see figure D)

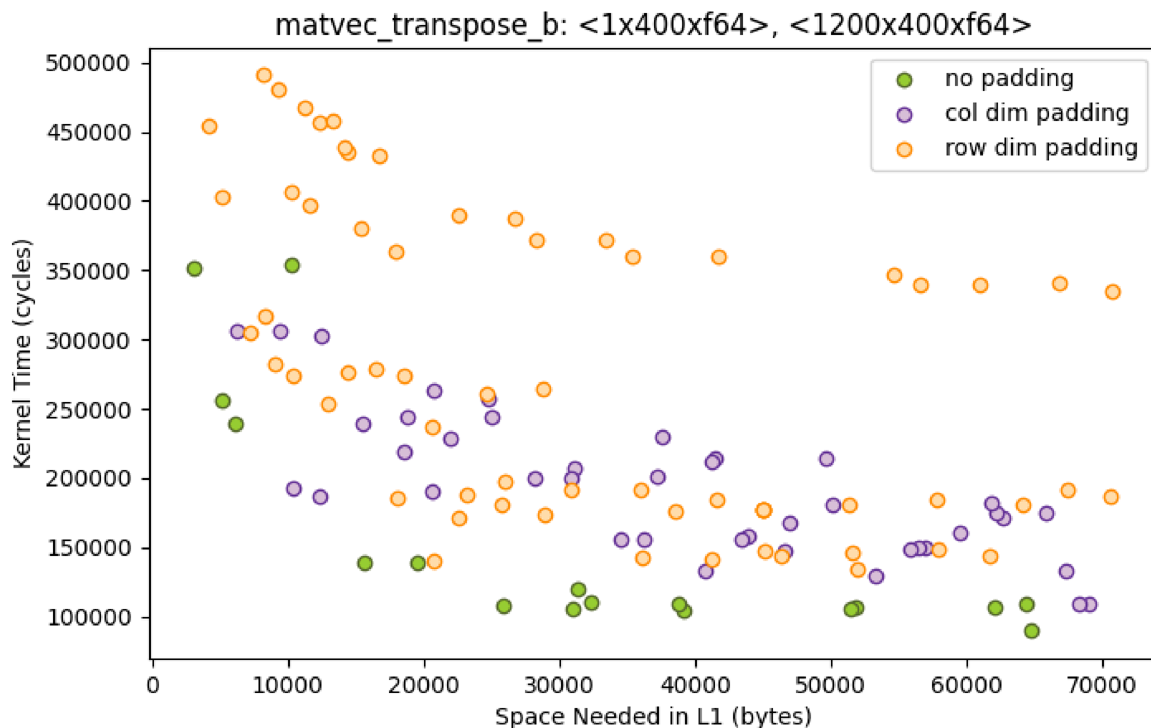


Figure B. Evaluating Tiling Schemes with and without padding for dispatch 1. Tiling Schemes without padding (green dots) outperform all other options.

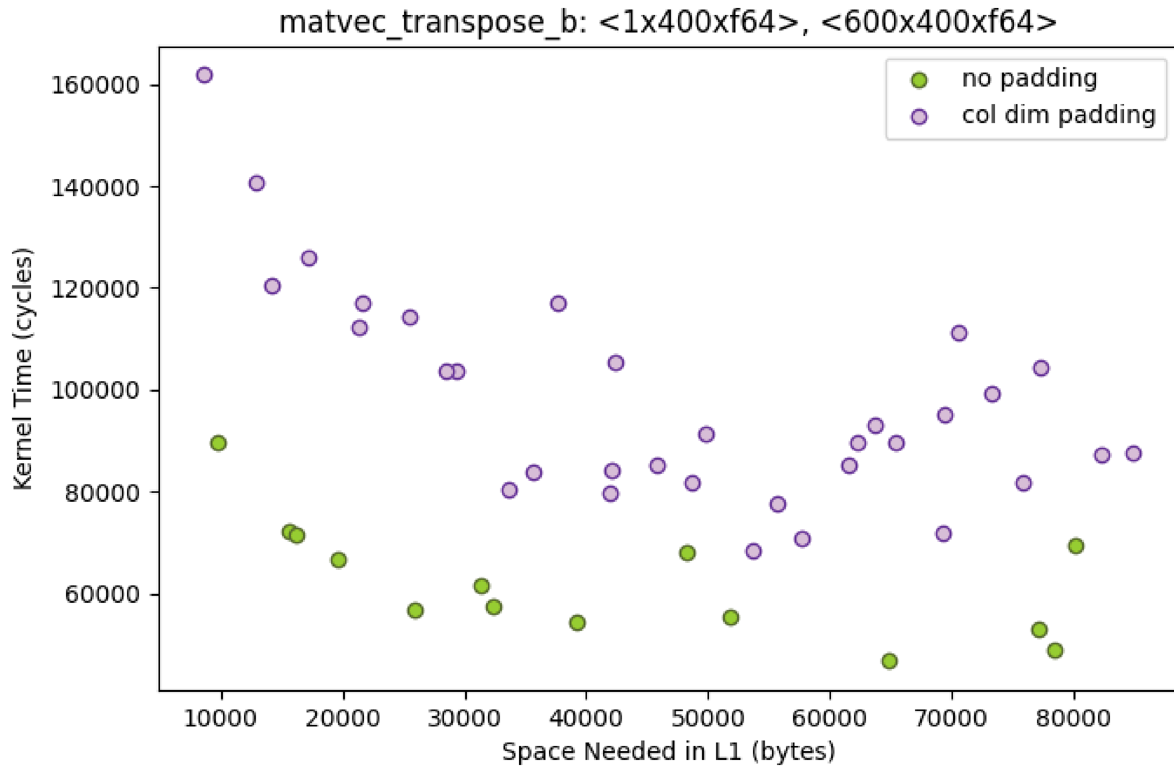
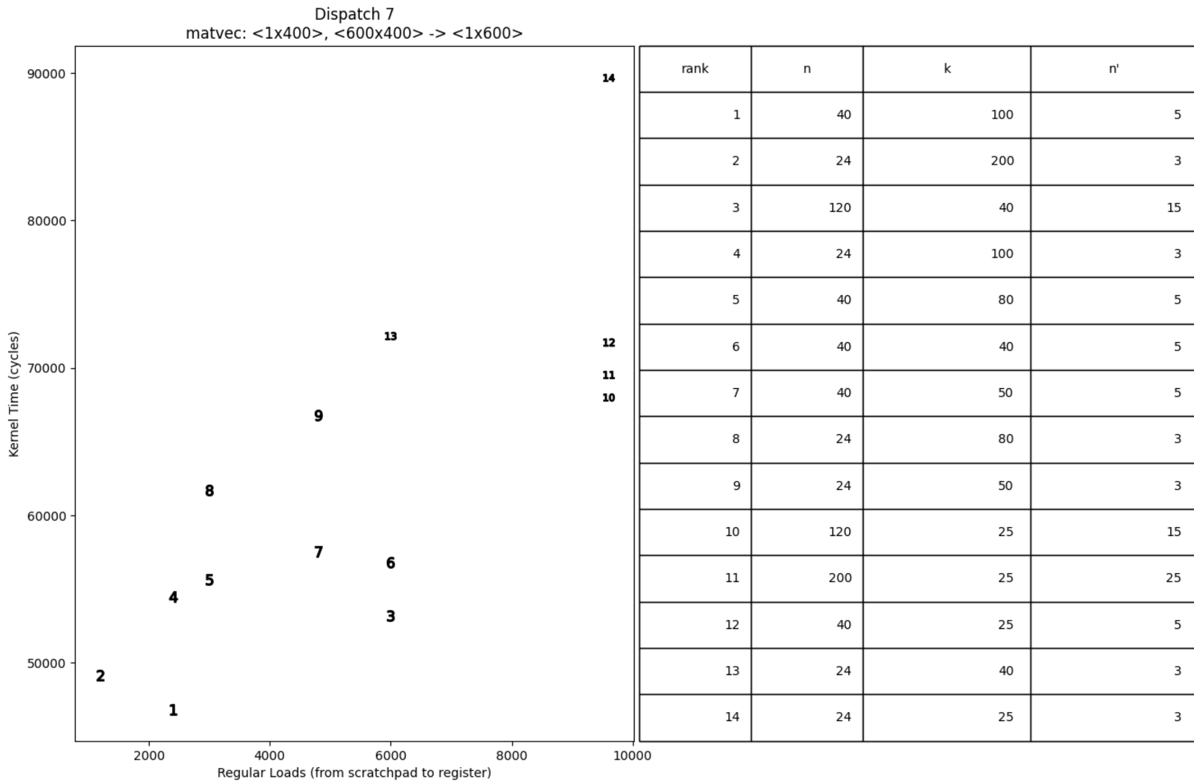


Figure C. Evaluating Tiling Schemes based on L1 usage (and padding) for dispatch 7. After filtering for no padding, "larger" tiling schemes generally out perform smaller ones.



rank	n	k	n'
1	40	100	5
2	24	200	3
3	120	40	15
4	24	100	3
5	40	80	5
6	40	40	5
7	40	50	5
8	24	80	3
9	24	50	3
10	120	25	15
11	200	25	25
12	40	25	5
13	24	40	3
14	24	25	3

Figure E. Evaluating Tiling Schemes based on the number of Regular Register Loads. In general, faster tiling schemes use fewer regular register loads (more streaming loads)

3.3.5.4 Analytical - Empirical Hybrid Model

The Myrtle Cost Model uses a combination of Algorithm Information, Compiler Lowering Information, Target Hardware Information, and Empirical Observations to select the optimal or near optimal tiling scheme for matmul transpose operations. We used **empirical observations**, such as the case of no padding, and **compiler lowering information**, such as the unroll and jam factor of xDSL's snitch hardware loop lowering, to prune our search space.

We used **target hardware information**, like the advantage of streaming registers (and disadvantage of streaming register configuration overhead) to identify potential factors contributing to tiled computation performance. **We use static analysis to quantify these factors for each tiling scheme.** Finally, we select the optimal or near optimal tiling scheme by filtering with an empirically determined priority.

3.4 SigSag

SigSag is an analytical modeling tool developed under Convolve at TUE for evaluating the energy consumption and latency of ultra-low power (ULP) spiking neural network (SNN) accelerator blocks at the system-on-chip (SoC) level. It provides performance analysis specifically tailored to the unique characteristics of SNNs, enabling hardware designers to estimate and optimize the efficiency of neuromorphic computing architectures. SigSag works in conjunction with other tools like ZigZag (for artificial neural networks) and Stream (for SoC-level modeling), helping explore design space and guide architectural decisions for edge AI hardware systems. This tool supports accurate design space exploration by modeling key hardware metrics such

as computation, memory, and scheduling for SNN ULP blocks within the context of workload and hardware configurations.

3.5 Instruction Fusion

Instruction fusion is a micro-architectural optimization that combines multiple architectural instructions (μ-op) into a single operation (Macro operation), improving performance by freeing up resources and improving performance (more work per μ-op). Most of the benefits are obtained by fusing memory operations, hence in what follows we describe our compiler support for fusion friendly code layout, with a focus on fusing memory operations.

3.5.1 State-of-the-art

3.5.1.1 ISA-Level Fusion

At the "low potential and hardware complexity" end of the spectrum, the ISA itself can provide fused instructions, such as **load** and **store** pair in ARMv8. In this case, the compiler is responsible for pairing two loads or two stores into a single instruction. High-performance designs may then implement this architectural instruction as a single μ-op to increase memory throughput. This scheme is cheap as it does not require the hardware to identify candidate pairs, but **requires the two accesses to be consecutive in the code, to have the same base register, and to access contiguous bytes in memory**. This often limits the potential to stack spills and fills. This technique does not comply with the RISC-V ISA as it breaks the requirement that the instructions feature a single destination register.

3.5.1.2 Exploiting the Cache Access Width with Non-Contiguous Accesses:

Cache data arrays are typically banked at the cacheline level, allowing fused memory instructions to access unaligned data across multiple banks using existing byte-enable and multiplexing logic—without requiring costly structural changes.
However, supporting such fusion may impact memory disambiguation logic. A clean approach is to use a base-aligned address with a one-hot byte mask, but this conflicts with traditional load/store queues (LQ/SQ) that are allocated at dispatch, as crossing cacheline boundaries would require multiple entries. While compatible with late-bound LQ/SQ designs, alternative solutions, such as storing multiple base addresses, incur added complexity and increase both comparisons and queue size.
To overcome this limitation we reorder instructions at the compiler level to convert non consecutive accesses to consecutive accesses thereby reducing the hardware complexity.

3.5.1.3 Speculative fusion for non-consecutive accesses

Many fusible instructions do not appear back-to-back in the binary, and therefore enabling non-consecutive fusion can boost performance, as in Helios[22], proposed by UMU. Following Helios terminology, the **nuclei** are the instructions to fuse and the **Catalyst** are the instructions in between the **nuclei**, which will either facilitate or prevent fusion. Unfortunately, having each instruction inspect “n” neighboring instructions in the allocation queue to determine if they can

be fused with distant instructions quickly becomes intractable. Therefore, Helios uses a predictive approach to identify combinations of non-consecutive, non-contiguous, and different base register (DBR) candidates.

3.5.1.4 Limitation of the State of the Art in Fusion:

A key limitation of speculative fusion is its reliance on complex hardware, which introduces high energy costs and potential rollbacks on misprediction. In contrast, our compiler-driven approach reorders instructions to convert non-consecutive memory accesses into consecutive ones without speculation. This eliminates misprediction overhead, reduces energy consumption, saves area, and achieves comparable performance with significantly lower hardware complexity.

3.5.2 Instruction reordering at LLVM IR for Instruction fusion

Context:
In LLVM, a **use-def chain** links each value definition to all its uses, enabling analyses like dead code elimination and instruction scheduling. These chains are efficiently represented thanks to LLVM's SSA form, where each variable is assigned exactly once. **Scalar Evolution (SCEV)** is another key analysis that models how scalar values evolve in loops using symbolic expressions, particularly through *add recurrence* (addrec) forms for induction variables. SCEV helps reason about loop bounds, pointer arithmetic, and variable values across iterations, making it essential for loop optimizations and alias analysis.

Contribution:
To enable instruction fusion in hardware, we perform selective reordering of memory operations, which requires precise dependence analysis. We identify fusible memory pairs using symbolic pointer reasoning from LLVM's Scalar Evolution (SCEV), ensuring they fall within a single cache line. To preserve correctness during reordering, we analyze instruction dependencies using LLVM's use-def chains and perform compile-time alias analysis to rule out unsafe memory interactions. We further integrate SVF to improve alias analysis coverage and incorporate graph-based interprocedural analysis, strengthening our ability to detect and exclude unsafe memory interactions across broader program contexts. These analyses allow us to safely hoist the tail instruction and its required computations without violating program semantics. Overall, our static analysis pipeline combines SCEV, use-def chains, and adapted alias analysis to support aggressive yet sound instruction reordering, enabling hardware-level fusion opportunities.

3.5.2.1 Validity of Reordering

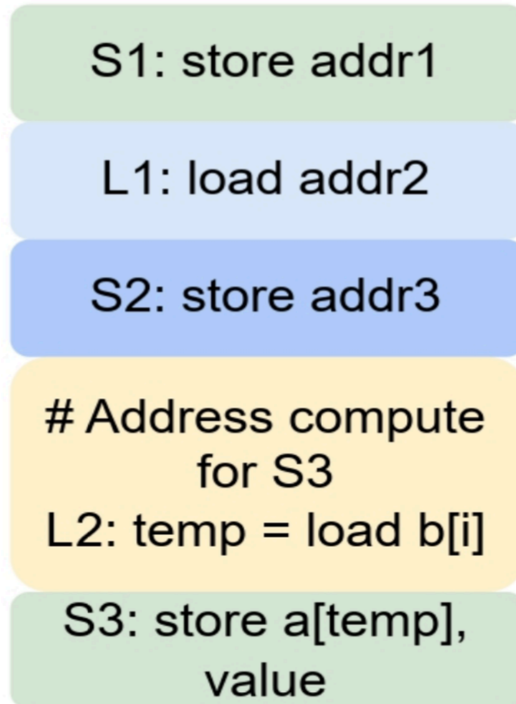


Figure 3.5.2.1 A: Memory dependency to reorder a Tail store. The fusible instructions are S1 and S3. The use-def chain (target address computation) of L3 contains L2. Our compiler verifies for potential conflicts between the loads (L2) from the use-def chain and other stores (S2) from the catalyst that proceed the usedef chain instructions and any conflicts with the Head store. Also, it verifies the Tail store (S3) does not conflict with any load (L1) or store (S2) from the catalyst it is reordered with or with the Head store. The checks are shown in the gray box. If no conflicts exist, reordering can proceed (right).

Context:
Pointer analysis is a static analysis technique used to determine what memory locations a pointer may reference during program execution. It is essential for reasoning about aliasing, enabling safe optimizations like loop transformations and instruction reordering. In LLVM, pointer analysis is conservative—it prioritizes correctness over precision by assuming aliasing when information is insufficient. While this ensures soundness, it can limit optimization potential. Despite this, pointer analysis remains critical for tasks such as memory disambiguation, escape analysis, and parallelization.

Contribution:
We perform memory dependence analysis to ensure correctness during reordering. The compiler verifies that the tail memory operation and its dependent instructions do not conflict with other memory accesses in the catalyst. This requires precise alias analysis across the tail's use-def chain and surrounding instructions. To handle function calls conservatively, we implement inter-procedural analysis along the call graph, eliminating standard compile-time limitations in alias analysis. For higher accuracy, we integrate and adapt a state-of-the-art alias analysis framework to fit our needs. These analyses allow the compiler to legally hoist the tail's

dependencies before the head and reorder the tail instruction, converting non-consecutive memory operations(ref Figure 3.5.2.1 A, S1 and S3 are non consecutive) into consecutive memory operations(S1 and S3 will be reordered as shown on the right ref Figure 3.5.2.1 B).

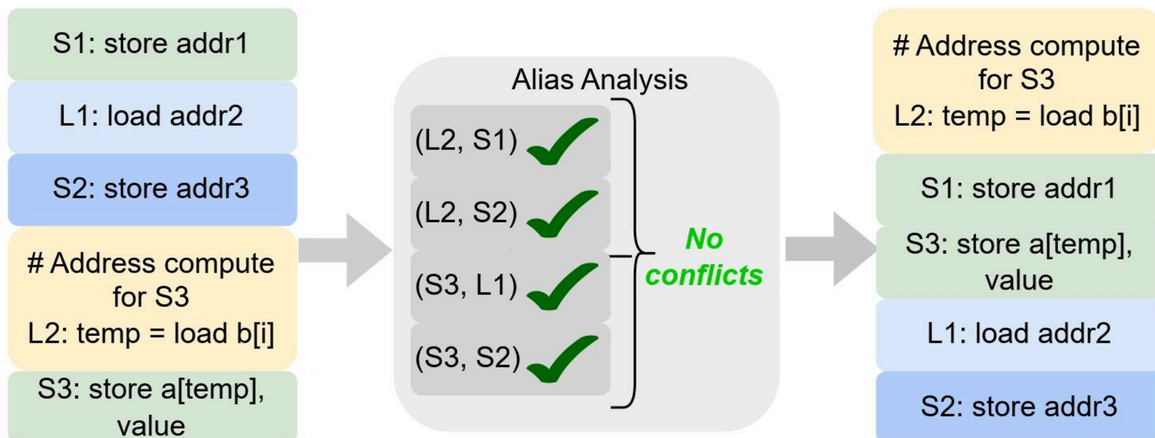


Figure 3.5.2.1 B : Memory dependency to reorder a Tail store. The fusible instructions are S1 and S3. The use-def chain (target address computation) of L3 contains L2. Our compiler verifies for potential conflicts between the loads (L2) from the use-def chain and other stores (S2) from the catalyst that proceed the usedef chain instructions and any conflicts with the Head store. Also, it verifies the Tail store (S3) does not conflict with any load (L1) or store (S2) from the catalyst it is reordered with or with the Head store. The checks are shown in the gray box. If no conflicts exist, reordering can proceed (right).

3.5.2.3 Increasing the Compiler's Reach.

The compiler conservatively reorders memory operations only within the boundaries of a basic block, which by design keeps register pressure at bay. However, to expand the window of instructions the compiler can operate on, CAIF enables dynamic loop unrolling, which is not included in the default O3 compiler optimizations. Dynamic loop unrolling (DLU) unrolls loops with statically unknown trip counts by guarding the unrolled iterations with conditionals. DLU can increase the number of live variables, potentially leading to register spilling. To limit this effect, we restrict DLU to innermost loops that have not been previously unrolled by O3, have less than 50 instructions, and contain at least one memory operation. Moreover, the unroll factor is 2 (and not higher).

3.5.2.4 Chained Alias Analysis using a database to reduce compilation-time

LLVM supports several alias analyses, each with distinct trade-offs between precision and coverage. For example, BasicAA efficiently analyzes pointer arithmetic but struggles with complex memory relationships; TBAA leverages type information but is less effective in low-level code; GlobalsAA targets global variables; and DSAA uses graph modeling to improve precision at the expense of scalability. While these analyses excel within their domains, individually they offer limited coverage. To address this, LLVM chains analyses: specialized results take precedence, and unresolved queries default to more conservative options. This cooperative approach increases both precision and coverage, enabling more effective optimizations.

Nevertheless, LLVM's native analyses remain conservative, sometimes limiting optimization opportunities. To improve this, we incorporate SVF (Static Value-Flow Analysis)[23] and GraphAA[14] . We fully integrated GraphAA to provide graph-based reasoning within LLVM. For SVF, which is only compatible with older LLVM versions, we created an integration layer: memory operations are instrumented with metadata and analyzed externally by SVF, with results stored in an SQL database and accessed by our LLVM pass. This enables efficient and scalable use of SVF"s advanced, flow-sensitive analysis within our pipeline.

Our mechanism allows seamless integration of state-of-the-art alias analyses from any LLVM version, eliminating the need to port them to the latest distribution. This approach enables leveraging the full range of analyses available as side LLVM projects. Importantly, it operates efficiently by using a fast SQL database to store and retrieve results, which significantly reduces compilation time (currently under review).

3.5.2.5 Limitation

The instruction order established in the compiler's middle-end is generally not preserved by the LLVM backend optimizations (llc -O3). To preserve the order of selected instructions in the final generated code, the following changes are necessary in the LLVM backend.

3.5.3 Backend Analyses and Transformations

To preserve the order at the LLVM backend, two techniques are employed.

3.5.3.1 Contribution: Extension of the DAGMutation

In the backend, LLVM IR is transformed into a directed acyclic graph (DAG), known in LLVM terminology as the **SelectionDAG**. Instruction selection, Instruction scheduling happen on DAG. Hardware can execute certain instructions efficiently when they are adjacent, such as compare and branch. To model such hardware constraints in the compiler, DAGMutation can be used. DAGMutation can add weak edges between schedule nodes, which can allow the instruction scheduler to prioritize nodes with weak edges over no edges. For example, in Figure 3.5.3.1, we want to schedule nodes A and B together, so we add a weak edge (Red edge) between A and B. In top-down scheduling, when node A is encountered, node B will be chosen next over C, due to the weak edge between (A,B). Add a weak edge (green edge) between B and C (shown in dotted line). In bottom-up scheduling, the weak edges ensure no instructions are scheduled between A and B. By marking C as dependent on B, C must be processed before B, keeping nodes B and A together. In both scenarios, the schedule order will be A, B, C, and D.

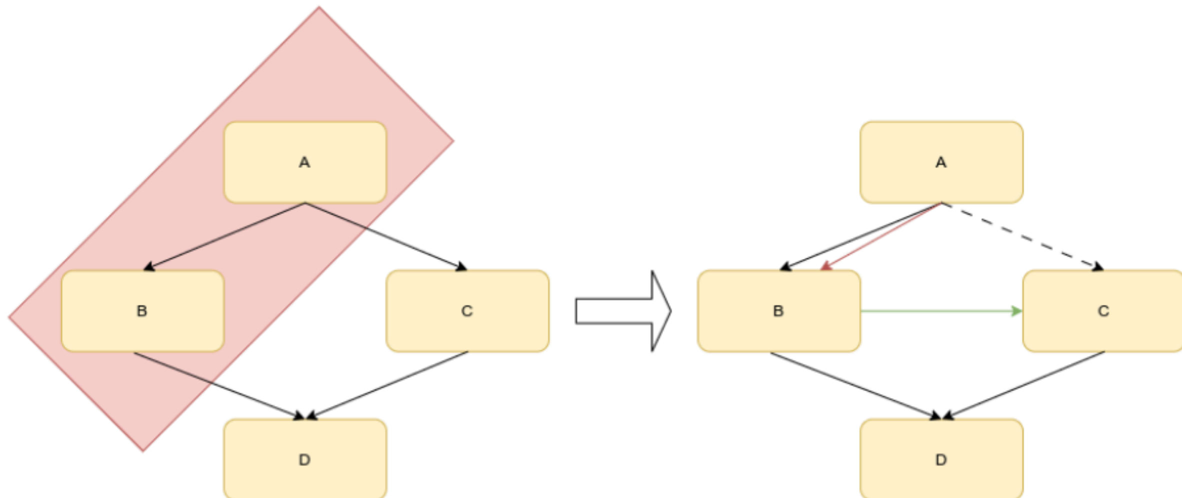


Figure 3.5.3.1 : Scheduling nodes A and B together

3.5.3.2 Drawbacks of DAGMutation

DAGMutation introduces weak dependencies that can be easily disrupted by register pressure. To address this, we complement DAGMutation with pseudo-instructions that impose stricter scheduling constraints. This approach balances the default scheduling heuristics with the order constraints required for fusion.

3.5.3.3 Pseudo Instruction

Pseudo-instructions are a more powerful mechanism to enforce clustering of memory operations. Through pseudo-instructions, two or more memory operations are packed into one, preventing the default scheduler and register allocator from breaking the desired order of the selected memory operations. We created new custom instructions for integer and floating-point pairs of loads and stores, respectively. Each pseudo-instruction is associated with a profile describing its operands and specific properties, which determine the behavior and constraints of these custom instructions. The original fusible instructions are replaced with the corresponding pseudo instructions, and the default backend optimizations are applied. However, O3 optimizations treat pseudo-instructions as single units, limiting individual instruction optimization and potentially causing redundant memory operations. A pseudo-instruction can only be removed if both memory instructions are redundant, which may hinder peephole optimization and dead code elimination. To address this, we remove unused nodes during instruction selection before replacing memory operation pairs with pseudo-instructions. Next, we employ a top-down scheduling policy, which minimizes register pressure. Since instructions are packed into pseudo-instructions, the order of the fusible memory operations is not impacted. Finally, after instruction scheduling and register allocation, pseudo instructions are expanded back into the original instructions, ensuring that Consecutive memory operations are maintained.

3.6 Periodic Scheduling of Layer-Fused CNN For A Compact Schedule Representation

CNNs are a dominant approach in computer vision tasks. An underlying reason for the increase in accuracy is the increase in size of the network, the layers and feature maps. Unfortunately, this has implications for the execution of CNN on edge devices, with on-chip memory of few megabytes, which might not fit the input image or its intermediate feature maps in its entirety, requiring spilling of intermediate computations to off-chip memory in a naive execution. The spilling of intermediate computation to off-chip memory is energy and bandwidth intensive, due to the sheer physical constraints of distance, noise and manufacturing variability that the signal traces require to overcome.

Layer-Fusion as an approach of reducing the energy intensive communication with off-chip

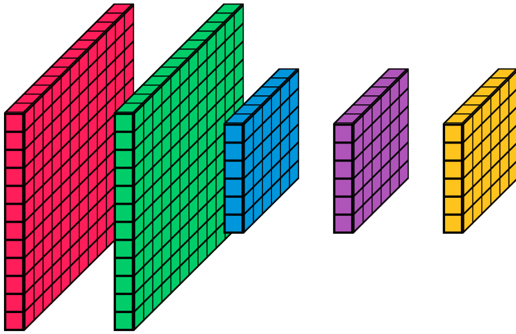
A highly explored and effective execution method of CNNs, in the field of computer architecture [4] [6], DSE [16] [24] and compiler engineering [27] [28] is "layer-fusion" (aka depth-first or cascaded execution), which consists in the interleaved execution of convolutional layers. The advantages of this method are two fold. Firstly, this method reduces the required memory of execution, allowing for a smaller on-chip memory footprint and fewer accesses to off-chip memory. Secondly, this method allows the hiding of both computation and data-transfers through pipelining, and maximizes parallelism and utilization of processors and communication channels.

While layer-fusion is an effective execution method, it makes efficient scheduling on heterogeneous architectures complex. The processing of multiple heterogeneous fused layers involves multiple hardware elements, such as a DMA for loading and storing of the intermediate activation from different memories, an AXI bus for transport of data, systolic arrays for the processing of convolutional layers, and many more specialized processing cores for convolution-like and element-wise operations, such as activation, max/average pooling, local normalization or element-wise addition, which may have their own specialized hardware accelerator or may be off-loaded to a general purpose processor due to lack of hardware acceleration. Furthermore, the frequent interleaved execution of convolutional layers makes a compact code generation non trivial, as a schedule requires a compact representation (as code) without affecting performance, in the form of latency or energy.

Example Network

As an example, we are going to consider the first 3 layers, made simple for the sake of explanation. The layers are, in order, a convolutional layer, max-pooling layer and another convolutional layer. We include the initial loading and storing, as they are detrimental parts for an actual implementation, which requires the loading of the input activation, and the storing of the output activation. Graphically, shown as the following Figure.

With the output tensor dimensions of each operation being the following.



The execution of such operation, is very commonly represented as nested for-loops. Following, are each operation represented as nested for-loops.

Load:

```
for oy in 0..12:
  for ox in 0..12:
    Load(O[oy][ox])
```

Conv1:

```
for oy in 0..12:
  for ox in 0..12:
    for fy in 0..3:
      for fx in 0..3:
        O[oy][ox] += I[c][7*oy + fy - 3][7*ox + fx - 3] * W[fy][fx]
        if fy == 2 && fx == 2:
          O[oy][ox] = max(0, O[oy][ox]) //relu
```

MaxPool:

```
for oy in 0..6:
  for ox in 0..6:
    for fy in 0..2:
      for fx in 0..2:
        O[oy][ox] = max(O[oy][ox], I[2*oy + fy - 1][2*ox + fx - 1])
```

Conv2:

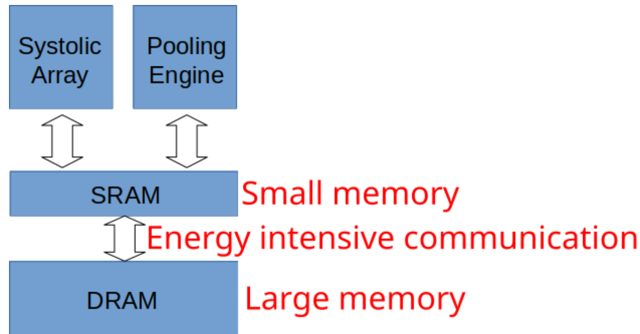
```
for oy in 0..6:
  for ox in 0..6:
    for fy in 0..3:
      for fx in 0..3:
        O[oy][ox] += W[c][oy + fy - 1][ox + fx - 1]*I[oy + fy][ox + fx]
```

Store:

```
for oy in 0..12:
  for ox in 0..12:
    Store(O[oy][ox])
```

Example Architecture and Motivation of Layer-Fusion

To illustrate the execution of such workload on hardware, we present the following architecture. It is composed of an off-chip memory (DRAM), an on-chip memory (SRAM), and two accelerators, a systolic array, which performs convolutional operation, and a pooling engine, which performs pooling operations.



The main motivation of layer-fusion is the avoiding of loading from DRAM to SRAM and off-loading from SRAM to DRAM, of the intermediate activations. This is because the communication with DRAM is usually energy and bandwidth intensive, due to the sheer physical constraints of distance, noise and manufacturing variability that the signal traces require to overcome. This is not made easy, due to the size of SRAM, usually beeing order of magnitude smaller than DRAM. A method of minimizing communication between SRAM and DRAM is layer-fusion, which we present in the next section.

Line-Based Layer-Fusion Execution

In a layer-by-layer execution, often, the entirety of the intermediate activation is materialized in memory, often requiring the spilling of the intermediate activation from SRAM to DRAM. To avoid the materialization of the intermediate tensors in its entirety, and to reduce the memory footprint and the spilling to DRAM, one approach is layer-fusion (aka depth-first execution or cascaded execution) [4] [6] [16] [24] [27] [28]. In this work we will focus on line-based layer-fusion [24] [7], which in a pseudo code form, is done by splitting the execution of each nested for-loop on the OY (or OX) dimension. In code form, this will be represented with `Load(oy)`, `Conv1(oy)`, `MaxPool(oy)`, `Conv2(oy)` and `Store(oy)` being the atomic units which outputs the `oy` line of the output tensor.
In code form, this becomes, for each operation:
Load:

```
for oy in 0..12:
-------------------------------------- Load(oy)
  for ox in 0..12:
    Load(O[oy][ox])
--------------------------------------
```

```
Conv1:
for oy in 0..12:
-------------------------------------- Conv1(oy)
  for ox in 0..12:
    for fy in 0..3:
      for fx in 0..3:
        O[oy][ox] += I[7*oy + fy - 3][7*ox + fx - 3] * W[fy][fx]
        if fy == 2 && fx == 2:
```

```
          O[oy][ox] = max(0, O[oy][ox]) //relu
--------------------------------------
```

Max-pool:

```
for oy in 0..6:
-------------------------------------- MaxPool(oy)
  for ox in 0..6:
    for fy in 0..2:
      for fx in 0..2:
        O[oy][ox] = max(O[oy][ox], I[2*oy + fy - 1][2*ox + fx - 1])
--------------------------------------
```

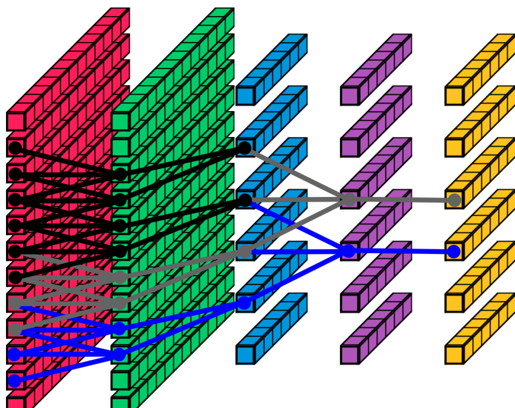
Conv2:

```
for oy in 0..6:
-------------------------------------- Conv2(oy)
  for ox in 0..6:
    for fy in 0..3:
      for fx in 0..3:
        O[oy][ox] += W[oy + fy - 1][ox + fx - 1]*I[oy + fy][ox + fx]
--------------------------------------
```

Store:

```
for oy in 0..12:
-------------------------------------- Store(oy)
  for ox in 0..12:
    Store(O[oy][ox])
--------------------------------------
```

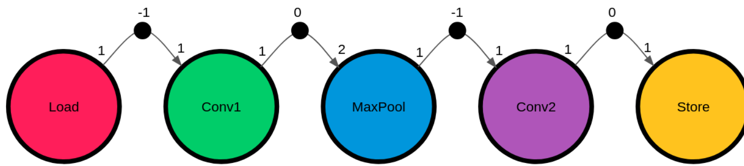
Graphically, line-based layer-fusion takes the following form, with each sub-tensors being the atomic unit of generation by each operation. The line represents the computational dependencies among tensors:



This introduces great complexity in code-generation, which, requires the scheduling of the execution of `Load(oy)`, `Conv1(oy)`, `MaxPool(oy)`, `Conv2(oy)` and `Store(oy)`, which requires the scheduling of respectively, 12, 12, 6, 6, 6 executions, or 42 in total. This is much worse for real world networks, such as ResNet18 [5], MobileNetV2 [21] , FSRCNN [3] and Xception [2] networks contain respectively 28, 52, 8 and 90 and convolutional, element-wise addition and max-pooling and layers, but in a line-based layer-fusion execution, requires the scheduling of 847, 1407, 6720, and 2928 elements. To allow for a compact code generation, we exploit the periodic dependence among tensors to compactly generate code, by scheduling a single period of a line-based layer-fusion execution.

Compact Representation of Line-Based Layer-Fusion Schedules

We represent the periodic data dependence as a Synchronous Dataflow Graph [13]. Given the example, such data dependence is represented in the following way:



To compactly represent the execution of line-based layer-fusion, we only schedule "one period" of line-based layer-fusion. The "one period" being the execution of additional sub-kernels which yield one output activation tensor line. In the following image, the blue squares contain the events which are scheduled and occur in one period. In (c) there is the same schedule executed in an ASAP manner, but for which, the order of the periodic execution is preserved.

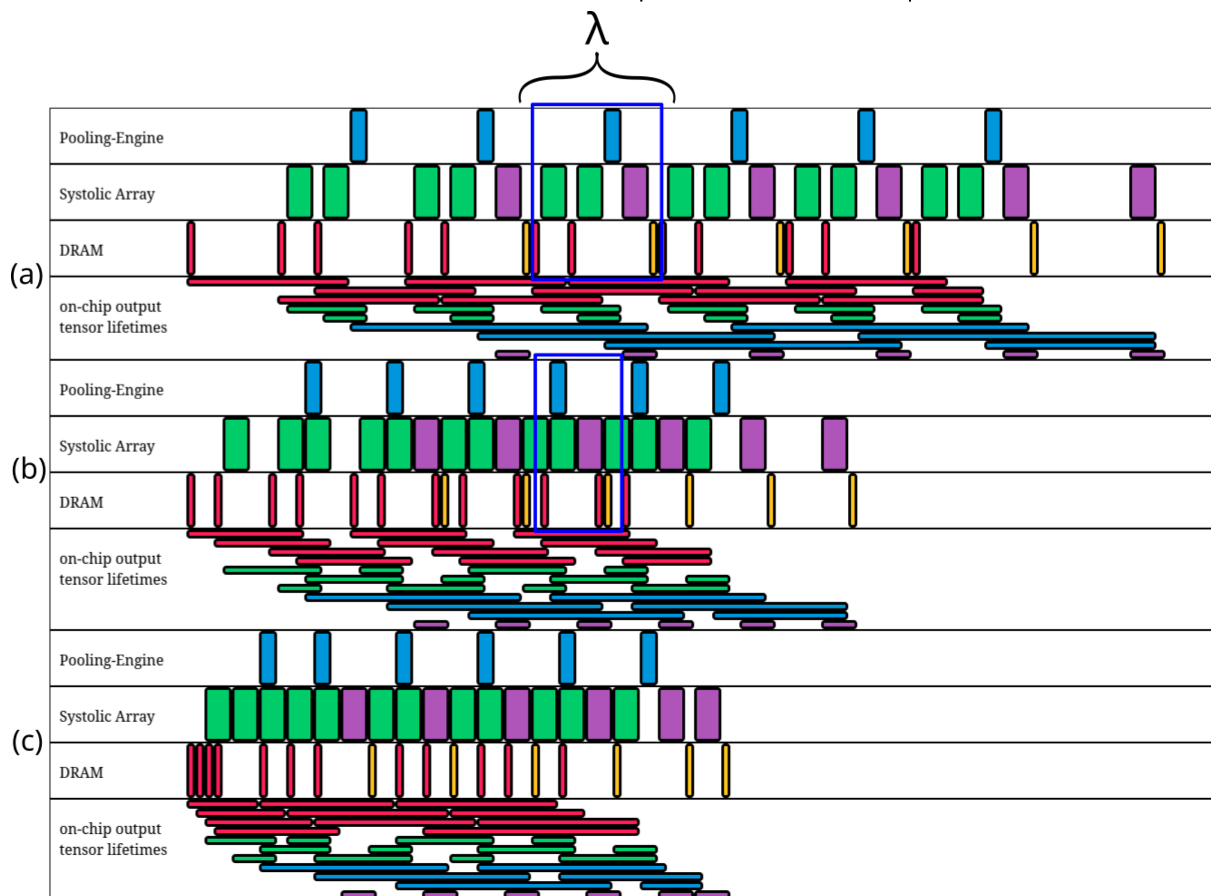


Figure 1: (a) The periodic execution of line-based layer-fusion of minimum on-chip memory occupancy. (b) The periodic execution of line-based layer-fusion of maximum throughput. (c) The ASAP execution which preserves the order of (b). In blue, is "one period" of the periodic schedules, where 1/λ is the throughput of the schedule.

In code form, the schedule of Figure 1c is encoded in the following pseudo-code, in which execution of 42 kernels is encoded into, in total, 7 kernels nested by for-loops.

Code executed by the Pooling Engine:

```
for i:
```

```
  MaxPool(i) if 0 ≤ i < 6
```

Code executed by the Systolic Array:

```
for i:
  Conv2(i - 3)  if 3 ≤ i < 6 + 3
  Conv1(2(i-1) + 1) if 1 ≤ i < 6 + 1
  Conv1(2i + 0) if 0 ≤ i < 6
```

Code executed by the DMA:

```
for i:
  Load(2i + 0) if 0 ≤ i < 6
  Store(i - 3) if 3 ≤ i < 6 + 3
  Load(2i + 1) if 0 ≤ i < 6
```

Comparison with SoTA

We use an ILP formulation to obtain schedules of maximum throughput while maintaining the memory occupancies of the SRAMs within their memory bounds. In our scheduling methodology, we allow spilling to DRAM only at stack splitting, like in the technique applied in [REF], with stack splitting heuristically obtained with at most 16 layers per stack. We implemented the schedules in the analytical model of Stream[REF] and evaluated our scheduling approach for evaluation of the quality of the schedules generated for line-based layer-fusion schedules. Tables 1 and Table 2 show that the adoption of this compressed schedule representation does not overly hinder the latency and energy of execution of CNNs. Table 3 shows that this schedule representation compresses schedules by <8%.

Latency [10^6 cc]	Stream (GA based mapping)	Ours (Mapping same as SOTA, with stack splittings includes at most 16 layers)
ResNet18	14.37	17.82
FSRCNN	38.62	38.63
MobileNetV2	3.63	3.34
Xception	71.28	66.70

Table 1: Latency comparison between Stream's default scheduler and our periodic scheduling methodology

Energy [mJ]	Stream	Ours
ResNet18	8.92	8.28
FSRCNN	10.82	10.81
MobileNetV2	1.29	1.11
Xception	29.03	29.39

Table 2: Energy comparison between Stream's default scheduler and our periodic scheduling methodology

Schedule Size	Total number of events	Compressed schedule representation (weight loading + events in a period)
ResNet18	1792	21 + 114
FSRCNN	10568	8 + 11
MobileNetV2	2600	42 + 151
Xception	6368	74 + 297

Table 3: Schedule size comparison, between total number of events(including loading and storing) and compressed schedule representation.

Conclusion and Future Work

We have shown that a periodic representation of the execution of layer-fusion greatly compresses the schedule representation without overly affecting latency and energy of execution. We observe that, for actual code-generation to occur, it is required the code generation of the kernel dispatch code, and the code generation of the kernel themselves. A MLIR based framework that handle the kernel dispatching functionality is Quidditch [30], which is further discussed in this work, and the the framework for the code-generation of each individual kernel could, is SNAX-MLIR [11], which uses ZigZag [17] to evaluate the latency and the energy that a spatial and temporal mapping provides.

3.7 Support for semi-automatic mixed-precision quantization

With this study, we intend to create an infrastructure capable of automatically generating quantized Python code starting from a Torch model or an ONNX file. The framework decomposes the neural network into a graph, similar to the previous "NeuralCasting" project. In this case, however, the generated data structure does not strictly follow the ONNX format but allows for arbitrary quantization. The framework first automatically generates Python code to calibrate the network. Subsequently, the generated JSON file for calibration contains the metadata to produce the Python code of the quantized model. The generated Python code is an entry point to emit an intermediate IR for an MLIR and LLVM-based compiler capable of generating assembly code, such as RISC-V architectures.

Currently, the framework can automatically generate code for a simple use case like an MLP for MNIST. However, with this work, we plan to compile and quantize more complex neural networks, such as a ConSeNet (Jabra's use case) for frequency domain denoising.

3.7 Integration of external libraries with the RISC-V lib ORT-SMD

ONNX Runtime (ORT) enhances model execution by applying a graph partitioner and optimizer that performs static transformations such as operator fusion, constant folding, and other performance-focused rewrites. After these optimizations, the graph is partitioned and each node or kernel is assigned to the most suitable Execution Provider (EP) hand-optimized libraries for specific hardware like RISC-V. These EPs replace standard ONNX operations with highly efficient, hardware-tailored implementations. When such an optimized model is imported into onnx-mlir, the EP-specific ops are treated as custom operations. To support such EP-only ops within the onnx-mlir toolchain, we extended onnx-mlir to explicitly lower these custom operations. At a high level, this involves first translating the EP ops into the ONNX-IR dialect, a high-level MLIR representation of ONNX ops. This step ensures that all necessary attributes and type information are preserved, allowing the ops to move correctly through the compilation pipeline. Following this, the custom ops are lowered into the internal runtime representation by allocating output buffers and constructing appropriate runtime calls that are compatible with the LLVM backend.

With this extended support, onnx-mlir can now incorporate the performance advantages of ORT's optimized EP implementations while still benefiting from MLIR-based transformations for core ONNX operations. This setup also lays the groundwork for dual-path execution, allowing developers to compare the performance of EP-optimized ops against the native onnx-mlir versions, potentially selecting the most efficient implementation dynamically.

3.2.5 Enhanced roofline model including configuration time (ACCFG)

ACCFG Roofline Model paper presents an advanced analytical model that extends the classical Roofline concept by incorporating detailed cache behavior, specifically focusing on fully associative caches. This model aims to provide more accurate predictions for the performance of memory-bound kernels by modeling cache miss rates and bandwidth limitations grounded in actual cache configuration parameters.

A key innovation in the ACCFG Roofline Model is its inclusion of configuration time—the overhead associated with compiling and setting up the transformations or optimizations. By factoring in this time, the model offers a more practical and realistic estimation of the achievable performance in real-world compiler scenarios.

Overall, the model helps compiler developers balance computational work and memory usage while taking into account both the kernel execution and the overhead of configuration, enabling more efficient code optimizations on modern architectures with complex cache designs. This integration of cache-aware modeling with configuration costs represents a significant step forward for predictive performance modeling in the compiler optimization domain.

4 Evaluation and Results

This section provides the results of the baseline compilation for a RISC-V target, followed by step by step evaluations of other optimizations developed in the consortium, presented in the previous section.

4.1 Baseline performance of neural networks compiled for x86 and RISC-V respectively

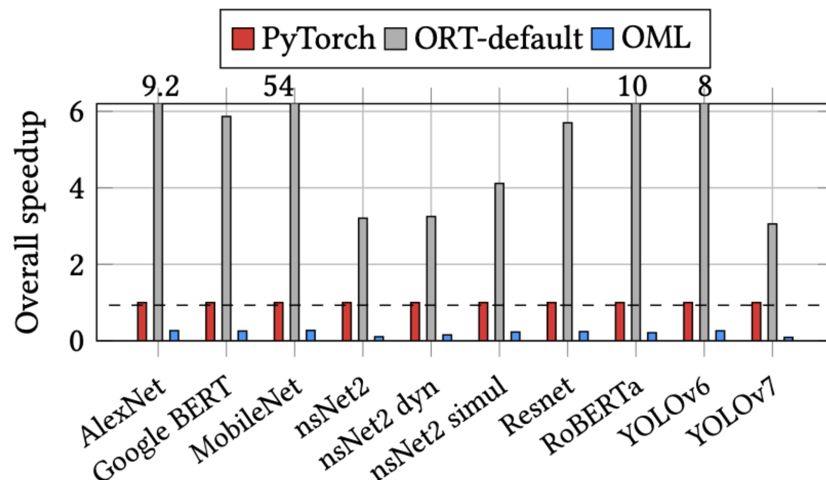


Figure 4.1 A: Speedup of ORT-default and OML (single-threaded) on x86 normalized to PyTorch

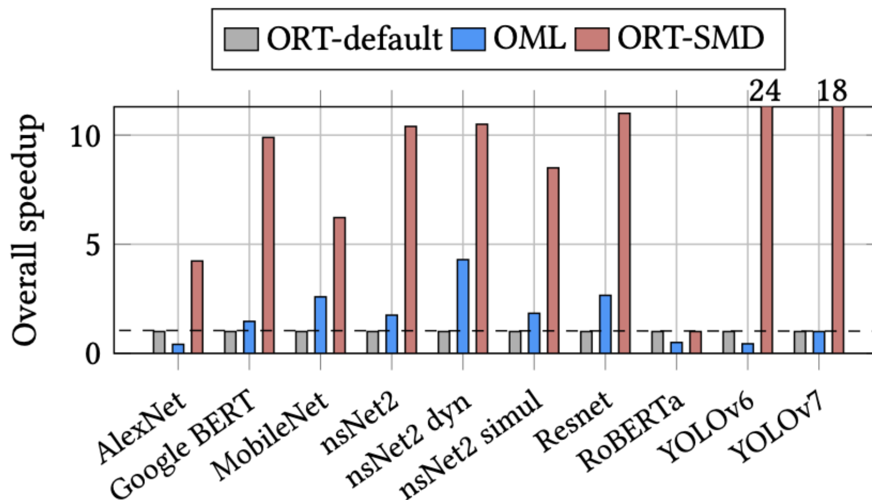


Figure 4.1 B: Speedup of ORT-default and OML (single-threaded) on x86 normalized to PyTorch

Figure 4.1A shows execution speedups for ten neural network models on an x86 Intel CPU, with results normalized to PyTorch. Each model was run ten times to ensure consistency. ORT-default consistently outperforms PyTorch and PyTorch JIT, with an average of 45% faster execution, thanks to hand-tuned libraries and hardware-specific Execution Providers. PyTorch JIT (not shown) offers up to 30% speedup for simpler models like AlexNet and MobileNet but

struggles with complex models such as BERT, RoBERTa, and nsNet2 due to dynamic input shapes.

Both PyTorch and ORT are multi-threaded by default, with Python front-ends and performance-critical C++ backends (e.g., libTorch, MKL-DNN). ORT includes low-level optimizations such as assembly-level kernels for specific operations like softmax. In contrast, our LLVM-based approach (OML) uses standard -O3 compiler optimizations at both IR and code generation stages, runs single-threaded, and does not rely on hand-optimized libraries—making it difficult to surpass the highly tuned PyTorch and ORT on x86
Figure 4.1B compares RISC-V performance for: (1) ORT, cross-compiled for RISC-V, and (2) OML, built using our compiler pipeline. Due to the unavailability of required libraries, PyTorch could not be evaluated on RISC-V.
On RISC-V, OML achieves performance comparable to ORT-default, despite the latter using hand-optimized libraries for select operations. This is largely because ORT-default lacks extensive fine-tuning for RISC-V, creating an opportunity for automated approaches to compete. OML leverages a fully automated compilation pipeline that exposes both MLIR and LLVM IR for high-level, target-agnostic optimizations, and utilizes the LLVM backend for hardware-specific tuning. While ORT relies on manually optimized kernels, OML's out-of-the-box compiler optimizations provide a scalable and portable solution for emerging platforms like RISC-V, often placing its performance between ORT-default and manually optimized ORT-SMD across a range of neural network workloads.

4.2 Performance analysis of NsNet

Model	GRU	Matmul	GRU speedup	Matmul speedup	overall speedup
nsNet2 baseline	2	4	1.4	1.2	1.7
nsNet2 dyn	2	4	1.8	1.3	4.2
nsNet2 simul	4	7	1.9	1.25	1.9

Table 4.2: Speedups summary: GRU and matmul in nsNet2 on RISC-V Atrevido 423 core

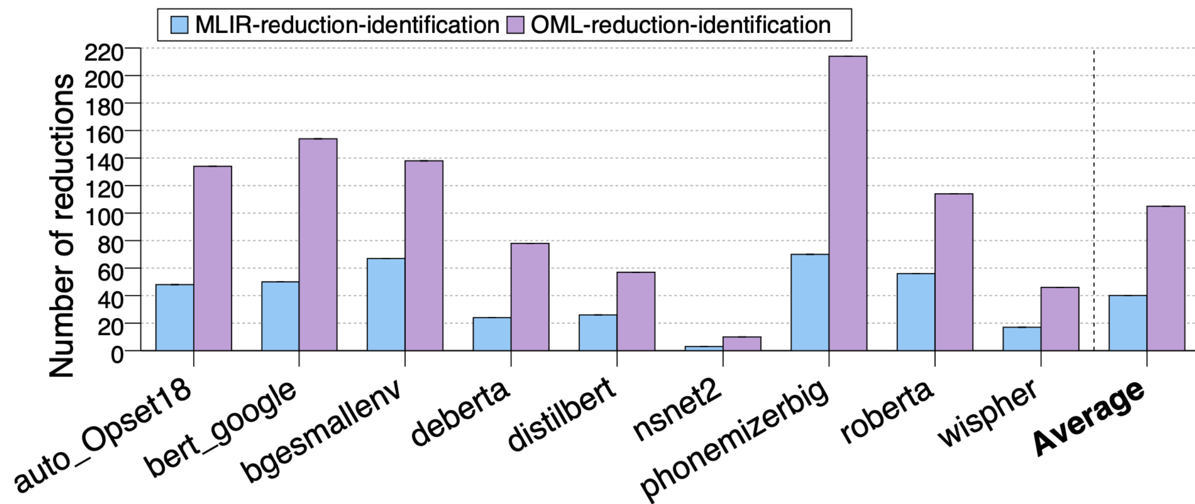
We evaluated three variants of nsNet2—a real-time denoising model for edge devices: (1) baseline, (2) dynamic layer-wise (nsNet2 dyn), and (3) dynamic simultaneous extra-layers (nsNet2 simul). On x86, profiling with PyTorch Profiler and Intel VTune showed 32 threads for the baseline and 35 for the dynamic versions, leveraging PyTorch's multi-threaded execution. No equivalent profiling tools were available for RISC-V.

LLVM vectorization reports for the baseline model showed that only 11 of 37 loops were vectorized, with common blockers being unsafe floating-point ops, optimal scalar cost, and unvectorizable calls.

On RISC-V Figure 2, OML outperformed ORT-default, achieving up to 4× speedup on nsNet2 dyn. The table above highlights GRU and MatMul layer counts, showing OML gains are tied to GRU-heavy workloads. Despite more GRU layers in nsNet2 simul, speedup was limited (1.9×) due

to ORT-default's optimized MatMul and GEMM. ORT-SMD's higher speedups highlight further optimization potential. Our goal is to integrate such tuning into OML automatically via LLVM.

4.3 Performance results with and without the manual opts for vectorization



Our proposed pass demonstrates significant improvements as shown in figure 4.9.1 A over the "match reduction test" pass for the following reasons:

1. The pass identifies reductions across multiple dialects, such as arith and math, whereas match reduction test works on arith dialect only
2. In Transformers and RNN (recurrent Neural Networks as nenet2, where matrix multiplication is a primary computational bottleneck, it is more efficient to use math.fma operations instead of separate mulf and addf operations. However, the MLIR match reduction test pass does not recognize the addition reduction present in math.fma, which adversely affects the vectorization of matrix multiplications. In contrast, our approach identifies the addition reduction present in math.fma as a reduction operation

In summary, our OML-reduction introduces a systematic decomposition strategy to identify and vectorize complex ternary reductions, including operations such as fused multiply-add (FMA). This fills a gap in MLIR, which lacks such reduction pattern detection. As a result, OML-reduction achieves up to 2.5× greater reduction coverage in NNs than default MLIR, especially benefiting auto-vectorization.

MLIR's out-of-the-box auto-vectorization fails to vectorize most of the MatMuls due to failure in reduction identifications and Therefore, there are no significant performance gains in ONNX-MLIR-no-custom-opts. The OML-reduction pass identifies more reductions MLIR's out-of-the-box auto-vectorization fails to vectorize most of the MatMuls due to failure in reduction identifications and Therefore, there are no significant performance gains in ONNX-MLIR-no-custom-opts. The OML-reduction pass identifies more reductions

than the original ONNX-MLIR-no-custom-opts flow, yet it does not yield performance gains when applied without data layout optimizations because the cost model (both MLIR and LLVM) predicts performance penalties due to non-contiguous data and hence it prevents vectorization. Hence, no performance improvements in OML. ORT includes a C/C++ library with manually vectorized code optimized for RISC-V. There is less fine-tuning for RISC-V for the ORT, thus OML-vect outperforms ORT on RISC-V for 5 out of 8 models. On average, auto-vectorization in OML-vect (OML-reduction + data layout optimizations) achieves 92% performance improvements over the baseline. These findings highlight the necessity of combining the reduction pass with data layout optimizations to fully leverage vectorization capabilities and achieve significant performance gains.

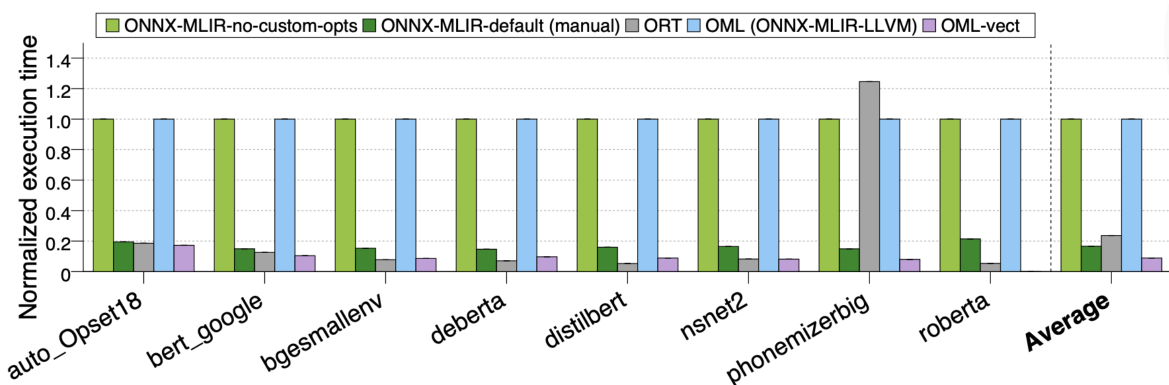


Figure 4.9.1 B: Performance on RISC-V

4.4 Performance results with Quidditch

4.1.2 Cost Model Driven Tiling

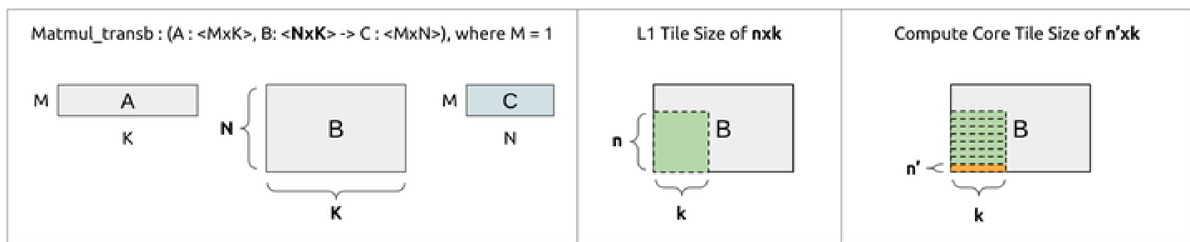
We evaluated UMU's proposal, Myrtle, on the Convolve use case NsNet2. For each viable Quidditch-compatible dispatch present in NsNet2, we compare the fastest observed tiling scheme with the tiling scheme selected by Myrtle.

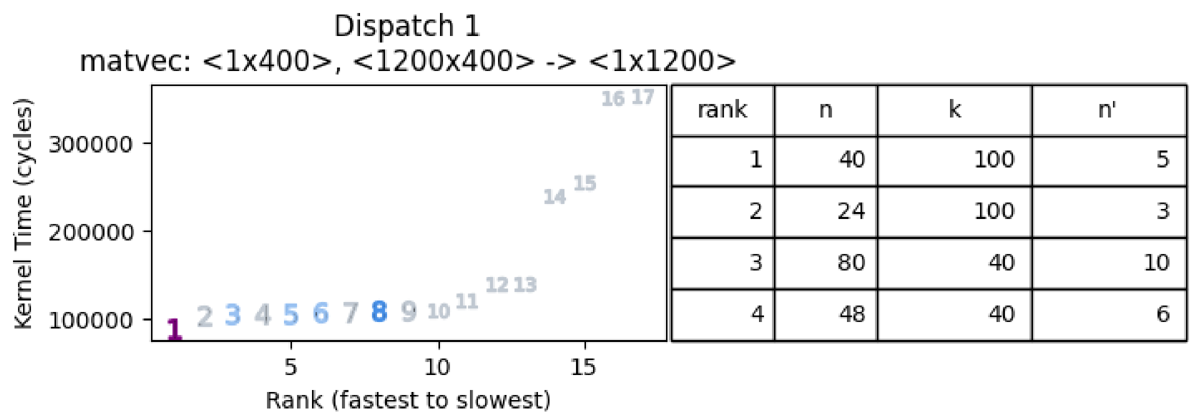
To measure cycle count of each dispatch within the larger NsNet2 run, we inserted a timing function before and after each dispatch at compile time. More specifically, when the code for the DMA core gets generated, we insert a "record_cycles" function before the first DMA transfer related to the dispatch and after the last DMA call related to the dispatch. The record_cycles function reads from the csrr register and stores the extracted cycle count in a designated position within a global array in L3. When NsNet2 completes, the contents of the array (cycle count for each dispatch) are printed to the console.

In figure A, for each NsNet2 dispatch, we graph the execution time of every tiling scheme in Myrtle's search space. The Y axis measures execution in cycles, while the X axis plots the points evenly from fastest to slowest using rank.

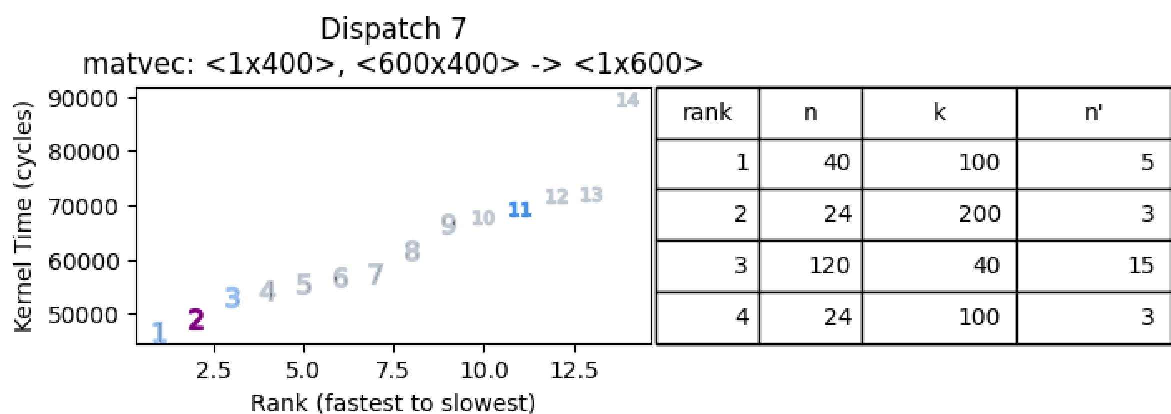
The progression of the marker colors from light to dark illustrates the progression of Myrtle's 3-step tile selection strategy:

- The lightest marker color, light gray, represents unselected search space points.
- Slightly darker in value is light blue, which represents search space points selected by the first step: SSR configuration count minimization.
- Darker still is a mid-range blue, which represents search space points selected by the second step, maximizing L1 usage.
- The darkest marker, purple in color, represents the result of the third step, picking the tiling scheme with the fewest number of regular loads.

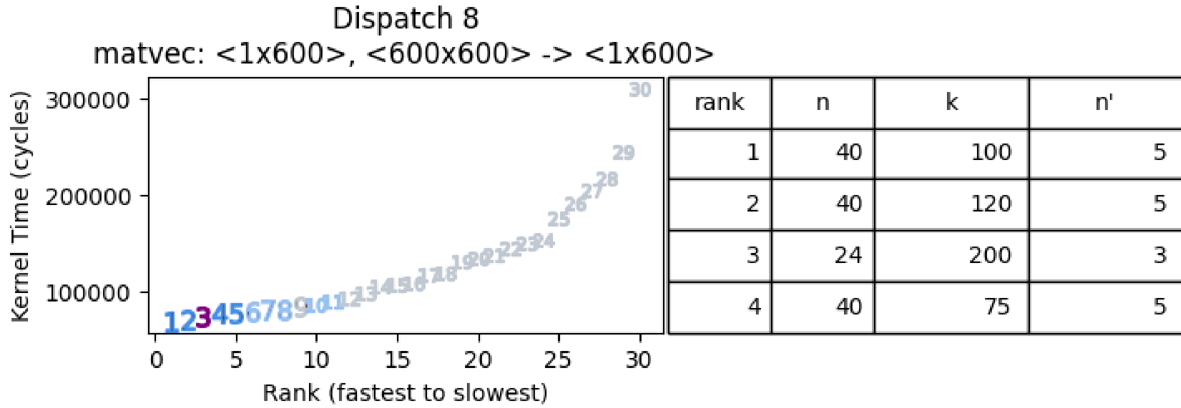




rank	n	k	n'
1	40	100	5
2	24	100	3
3	80	40	10
4	48	40	6



rank	n	k	n'
1	40	100	5
2	24	200	3
3	120	40	15
4	24	100	3



rank	n	k	n'
1	40	100	5
2	40	120	5
3	24	200	3
4	40	75	5

Figure 4.1.A: Myrtle Tiling Scheme Selection using Flat Filtering.
Light blue minimizes the number of microkernel runs, **dark blue** maximizes L1 usage, and **purple** (final pick) minimizes the number of regular loads from scratchpad to register (indirectly maximizing streaming register loads).
The rank of the purple points are 1, 2, and 3 respectively, indicating that Myrtle picks the best possible tile for dispatch 1, the second best tiling scheme for dispatch 7, and the third best tiling scheme for dispatch 8. But how close in execution time are the second and third best tiling schemes from the best tiling schemes?

For vector matrix transpose operations within NsNet2, Myrtle automatically selects a tiling scheme within 5% of the fastest observed tile.

Dispatch	Fastest Observed Tiling Scheme : cycles	Myrtle Selected Tiling Scheme : cycles	Fastest Observed / Myrtle Selected
0	1-200-23: 29420 cycles	1-200-23: 29420 cycles	1.00
1	1-40-100: 89889 cycles	1-40-100: 89889 cycles	1.00
7	1-40-100: 47182 cycles	1-24-200: 49157 cycles	0.96
8	1-40-100: 69685 cycles	1-24-200: 73069 cycles	0.95

4.5 Performance results on Instruction reordering and fusion

This section evaluates the impact of compiler assistance on instruction fusion, and investigates:

(1) the degree to which CAIF shifts fusion from Base (-O3) to UMU's proposal, CAIF (Compiler assisted instruction fusion) Figure 4.5.A;

(2) performance with varying hardware complexity

Our out-of-order core model has been extended to implement consecutive instruction fusion. In particular, we examined several configurations of the hardware with different complexity requirements, for each hardware configuration we run both the baseline compiler and our compiler support (CAIF):

1. **NoFusion**: Fusion is disabled in the hardware. This represents our baseline in the figure below.
2. **ConsSBR**: This hardware is able to fuse only Consecutive-SBR-contiguous instruction pairs. It is a non-speculative fusion implementation. First two bars in the figure below.
3. **ConsSBR64**: This hardware is able to fuse both Consecutive-SBR-contiguous and Consecutive-SBR-64bytes instruction pairs. It is also a non-speculative implementation. This configuration is used for the 3rd and 4th bar below.
4. **Cons**: This hardware is able to fuse all consecutive instruction pairs. It requires speculative support. This configuration is used for the 5th and 6th bar below.
5. **Helios**: This state-of-the-art hardware enables all kinds of fusion, including non-consecutive pairs. It is also a speculative solution. This configuration is used for the 7th and 8th bar below.

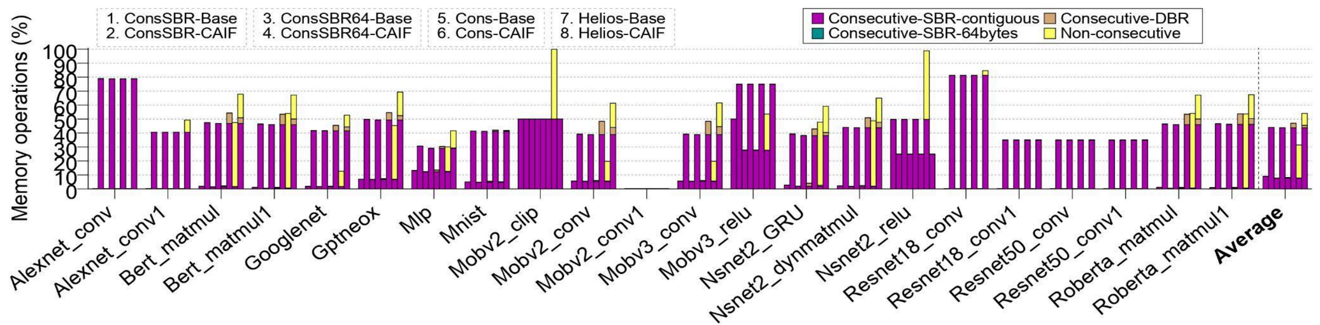


Figure 4.5.A: Instruction fused with varying hardware complexity

With compiler reordering, CAIF increases the average value of Consecutive-SBR-contiguous from 7.76% to 43.72%. The process of fusing consecutive memory operations can be achieved with relatively simple hardware.

When examining ConsSBR64 hardware, we observe that (i) CAIF increases the percentage of fused instructions by nearly six times (as shown by the third and fourth bars in the figure, and (ii) the proportion of fused instructions with CAIF on ConsSBR64 (fourth bar, 65.01%) is more than 30% higher than that achieved by state-of-the-art Helios running code from the base compiler (seventh bar, 34.28%).

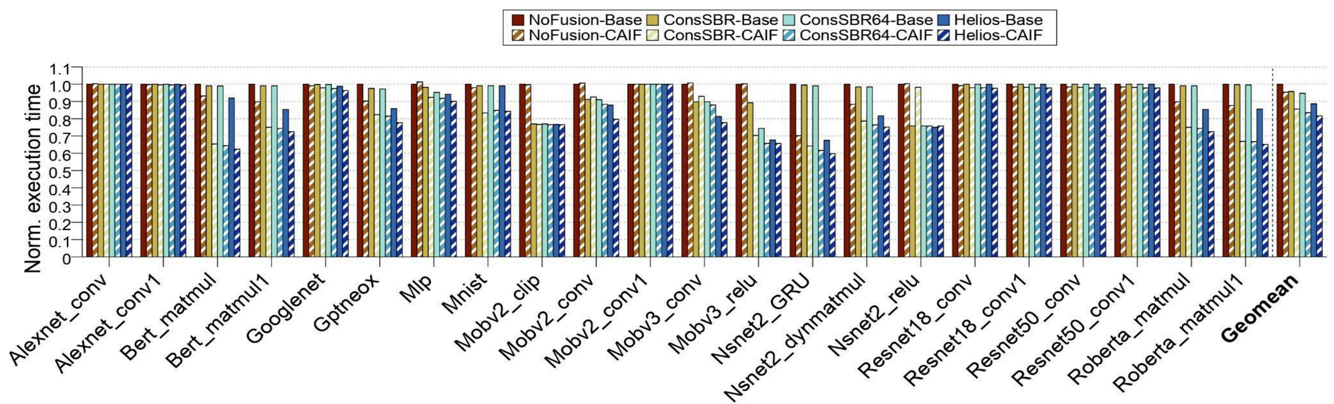


Figure 4.5.B Execution time with varying hardware complexity

Execution time is normalized to compiler baseline (-O3) with hardware fusion disabled (No-Fusion). When comparing CAIF on No-Fusion hardware to the Base configuration, CAIF introduces no additional overhead and does not increase register pressure. With the simplest fusion hardware (ConsSBR), CAIF reduces average execution time by 14.4% compared to NoFusion-Base, whereas the baseline compiler achieves a 4.2% reduction. CAIF also demonstrates strong performance when both non-speculative fusion techniques are used (ConsSBR64), reducing execution time by approximately 16.6% relative to NoFusion-Base, while the baseline compiler achieves a 5.3% reduction. These results indicate that instruction reordering with CAIF significantly enhances the effectiveness of consecutive instruction fusion, even on simple hardware.

4.6 Performance results on Instruction reordering and fusion for general purpose benchmarks

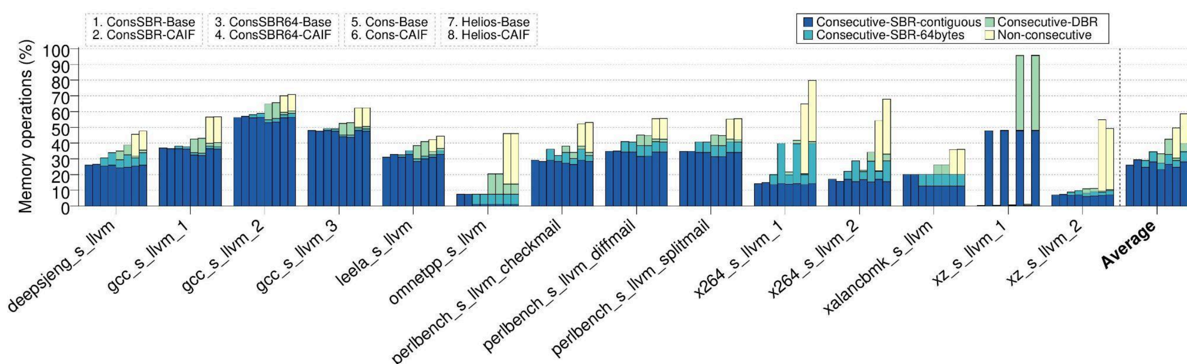


Fig 4.6 A, Instruction fused in general purpose applications, with varying hardware complexity

With compiler reordering, CAIF increases the average value of Consecutive-SBR-contiguous from 22.64% to 29.38%. The process of fusing consecutive memory operations can be achieved with relatively simple hardware.

When examining ConsSBR64 hardware, we observe that (i) CAIF increases the percentage of fused instructions by 54% (as shown by the third and fourth bars in fig.4.6, and (ii) the proportion of fused instructions with CAIF on ConsSBR64 (fourth bar, 37%) which is only 9% less than that achieved by Helios running code from the base compiler (seventh bar, 44.76%).

With Cons i) with CAIF(Cons-CAIF 45.33% 6th bar) finds same number of fusion pairs as helios-base (44.76%7th bar)

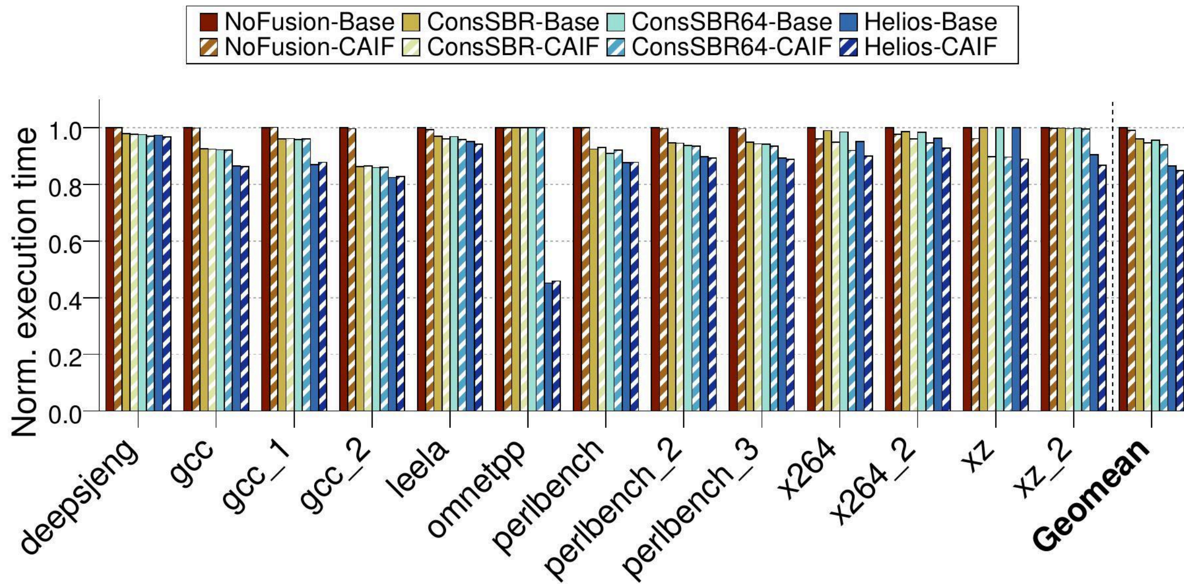


Fig 4.6 B, Execution time with varying hardware complexity

Execution time is normalized to compiler baseline (-O3) with hardware fusion disabled (No-Fusion). When comparing CAIF on No-Fusion hardware to the Base configuration, CAIF introduces no additional overhead. With the simplest fusion hardware (ConsSBR), CAIF reduces average execution time by 5.4% compared to NoFusion-Base, whereas the baseline compiler achieves a 4% reduction. CAIF also demonstrates strong performance when both non-speculative fusion techniques are used (ConsSBR64), reducing execution time by approximately 6.1% relative to NoFusion-Base, while the baseline compiler achieves a 4.5% reduction. These results indicate that instruction reordering with CAIF significantly enhances the effectiveness of consecutive instruction fusion, even on simple hardware.

For SPEC, state of the art SLP vectorization is SNSLP[19], which gains max 2% and an average of less than 0.5% over LSLP[20] (LSLP performs same as -O3 on average, LSLP degrades performance in some benchmarks) both SNSLP and LSLP are evaluated on SPEC2006 with all other vectorizers disabled. SNSLP degrades performance in some benchmarks, whereas UMU's proposal on instruction reordering for fusion never hurts performance, not even when reordering is done without fusing. ConsSBR64-CAIF gains 6.1% over NoFusion-Base whereas ConsSBR64-Base gains 4.5% over NoFusion-Base.

4.7 Performance results of integrating external libraries

4.7.1 Optimizations for NNs for RISC-V

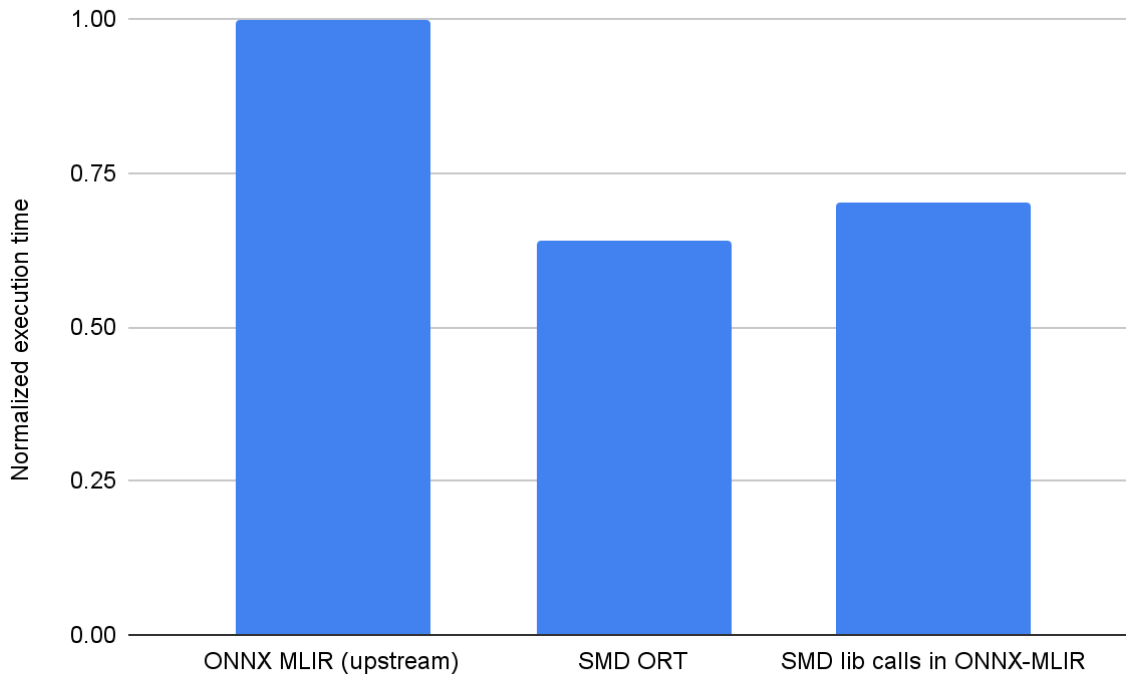


Figure 4.7.1: Performance of integrating external libraries in ONNX-MLIR flow

Figure 4.7.1 presents our preliminary evaluation of integrating manually optimized external library calls for specialized kernels such as matrix multiplication and convolution. Offloading these key compute-intensive kernels to highly optimized external libraries significantly boosts performance, since such libraries are often hand-tuned for specific architectures and therefore achieve better efficiency than the generic upstream ONNX-MLIR implementations. This strategy yields a reduction in execution time of more than 25% compared to the upstream baseline, demonstrating the substantial benefits of external implementations for computationally demanding operations. At the same time, we observe a 5% overhead when making these external calls within ONNX-MLIR. The source of this overhead lies in the fact that the functions are not inlined, and MLIR introduces additional function wrappers around the calls. While this overhead slightly reduces the potential performance gains, the overall effect remains strongly positive, indicating that the benefits of external libraries substantially outweigh the costs. These results also suggest that further improvements, such as reducing wrapper overhead or enabling inlining, could unlock even greater performance benefits.

4.7.2 Quantization

In this project, we aim to build a framework that can automatically generate MLIR code for mixed-precision quantized neural networks starting from PyTorch or ONNX models. Mixed-precision quantization assigns different bit depths to different tensors, since not all tensors react in the same way to perturbations or value approximations. This makes it more effective than uniform quantization, where every tensor is forced to the same precision, although at the cost of higher complexity.

Our objective is to produce an intermediate representation (IR) in MLIR, implemented using the Affine dialect. We chose MLIR because existing frameworks and runtime engines, such as ONNX Runtime and TFLite, provide only limited support for quantization, typically restricted to 8-bit uniform quantization. TFLite does allow some form of mixed-precision quantization, but its granularity is low, usually limited to setting separate precisions for weights and activations. Instead, MLIR allows for greater freedom in using different types of data.

At the current stage, we are working on defining the individual mixed-precision kernels directly in MLIR.

4.8 Performance results with Quidditch

4.8.1 Cost Model Driven Tiling

We evaluated UMU's proposal, Myrtle, on the Convolve use case NsNet2. For each viable Quidditch-compatible dispatch present in NsNet2, we compare the fastest observed tiling scheme with the tiling scheme selected by Myrtle.

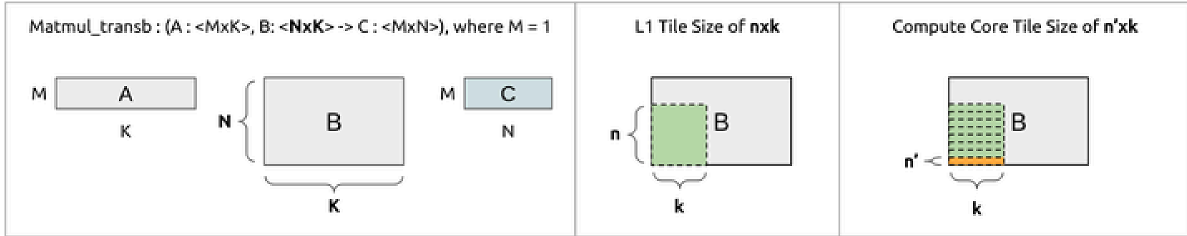
To measure cycle count of each dispatch within the larger NsNet2 run, we inserted a timing function before and after each dispatch at compile time. More specifically, when the code for the DMA core gets generated, we insert a "record_cycles" function before the first DMA transfer related to the dispatch and after the last DMA call related to the dispatch. The record_cycles function reads from the csrr register and stores the extracted cycle count in a designated position within a global array in L3. When NsNet2 completes, the contents of the array (cycle count for each dispatch) are printed to the console.

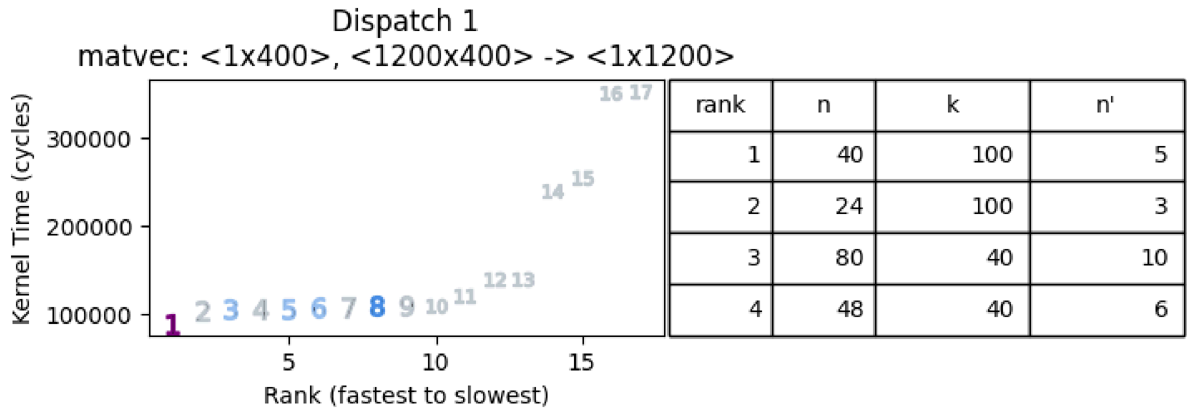
In figure A, for each NsNet2 dispatch, we graph the execution time of every tiling scheme in Myrtle's search space. The Y axis measures execution in cycles, while the X axis plots the points evenly from fastest to slowest using rank.

The progression of the marker colors from light to dark illustrates the progression of Myrtle's 3-step tile selection strategy:

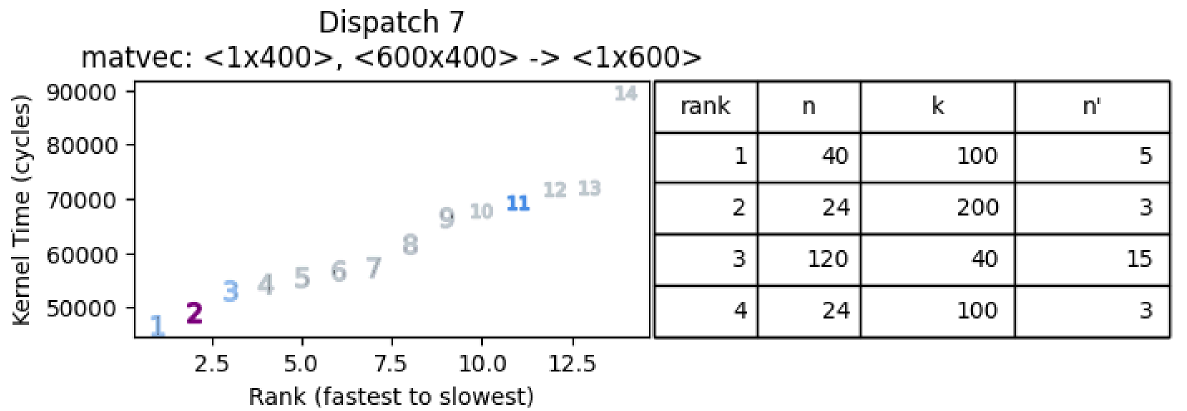
- The lightest marker color, light gray, represents unselected search space points.
- Slightly darker in value is light blue, which represents search space points selected by the first step: SSR configuration count minimization.
- Darker still is a mid-range blue, which represents search space points selected by the second step, maximizing L1 usage.

- The darkest marker, purple in color, represents the result of the third step, picking the tiling scheme with the fewest number of regular loads.

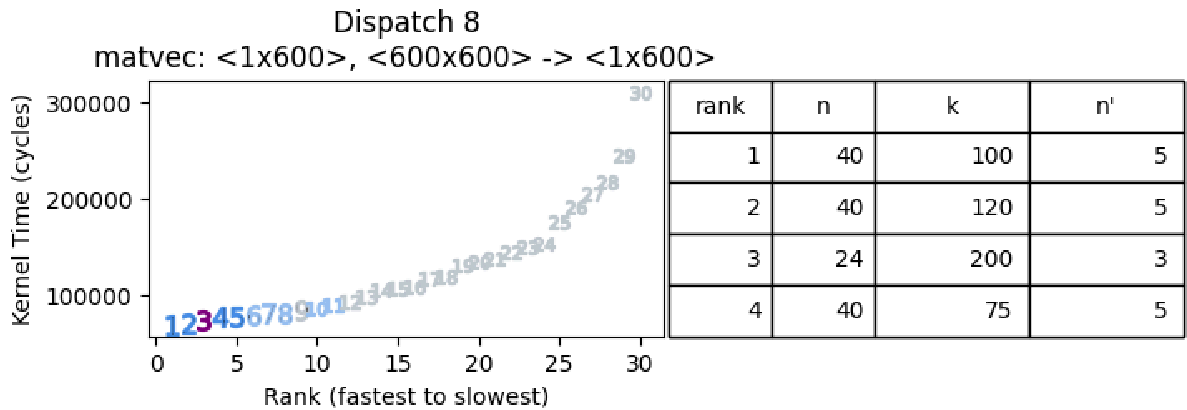




rank	n	k	n'
1	40	100	5
2	24	100	3
3	80	40	10
4	48	40	6



rank	n	k	n'
1	40	100	5
2	24	200	3
3	120	40	15
4	24	100	3



rank	n	k	n'
1	40	100	5
2	40	120	5
3	24	200	3
4	40	75	5

Figure 4.8.1: Myrtle Tiling Scheme Selection using Flat Filtering. **Light blue** minimizes the number of microkernel runs, **dark blue** maximizes L1 usage, and **purple** (final pick) minimizes

the number of regular loads from scratchpad to register (indirectly maximizing streaming register loads).
The rank of the purple points are 1, 2, and 3 respectively, indicating that Myrtle picks the best possible tile for dispatch 1, the second best tiling scheme for dispatch 7, and the third best tiling scheme for dispatch 8. But how close in execution time are the second and third best tiling schemes from the best tiling schemes?

For vector matrix transpose operations within NsNet2*, Myrtle automatically selects a tiling scheme within 5% of the fastest observed tile.

Dispatch	Fastest Observed Tiling Scheme	Myrtle Selected Tiling Scheme	Fastest Observed / Myrtle Selected
0	1-200-23: 29420 cycles	1-200-23: 29420 cycles	1.00
1	1-40-100: 89889 cycles	1-40-100: 89889 cycles	1.00
7	1-40-100: 47182 cycles	1-24-200: 49157 cycles	0.96
8	1-40-100: 69685 cycles	1-24-200: 73069 cycles	0.95

*Due to a code generation bug in Quidditch farther down in the pipeline (which causes certain tiling schemes to fail compilation), we are unable to report on results for dispatch 9 at this time.

5 Activities within CONVOLVE

- Disseminate UMU's contributions on static analyses and optimizations, developed under Convolve at:
    - CODAI talk, Best Paper Award: Insights into Interpreters, Compilers, and Optimizers for Neural Networks
    - Invited talk at REACH 2024: *Optimizing Code Layout: Enhancing Performance and Efficiency through Instruction Reordering*
- Organized the Convolve Plenary meeting in Murcia, at UMU, in February 2025
- UMU hosted GNU intern, Alessandro Cerioli, working on automating mixed quantization in UMU's compiler flow (from 2nd of October 2024 to 21st of December 2024).

- Hosted a number of external interns that worked on Convole compiler technologies (support for transformers and general purpose code optimization)
- Research visits at various Convolve partners:
    - Convolve WP5 Hackathon: April 2024
      PhD student Emily Sillars from UMU attended the Convolve WP5 Hackathon in Cambridge from the dates of Monday, April 1st, through Friday, April 5th. Work completed includes the initial development of a Linalg MLIR to ZigZag workload object conversion tool, in collaboration with CONVOLVE team members Joren Dumoulin and Chris Vasiladiotis.
    - Master student Ravikiran attended KU Leuven Convolve training, 22-24 May 2024. The first day of the training was about Zigzag/Stream, the second day was about the SNAX accelerator and the third day was about GVSOC. This internal workshop provided insights into Convolve hardware and how the Zigzag helps with mapping data to the accelerator
    - PhD students Shreya and Ravikiran visited Cambridge for a research visit, working with Linalg from MLIR, and with tools developed by our partner: Xdsl and Quidditch. This visit was aimed to help integrate the UMU passes in the Quidditch compiler.

6 Conclusions

UMU studied the optimizations offered by state-of-the-art frameworks such as PyTorch and ORT. We highlighted the significant performance gap between manually fine-tuned libraries and the proposed pipeline OML on x86 architectures. Our findings indicate that neural network public libraries for RISC-V are very immature, with the proposed OML mostly outperforming the default implementations. While manual fine-tuning shows promise for optimizing performance on RISC-V, the process is labor-intensive and not scalable. Therefore, we automate the most promising of these optimizations through the MLIR and LLVM compiler, bridging the gap and smooth integration of neural network inference on RISC-V architectures. The RISC-V is the underlying platform for the Convolve accelerators. We also provide a path to integrate emerging external libraries that optimize neural network kernels for RISC-V, in our OML compilation flow.
To analyze the Convolve use-cases (neural networks), as well as general-purpose code, we built powerful chained alias analyses and kept the results in fast-to-query databases, reducing compilation time significantly, after a warm-up phase to populate the database. With the help of this analysis, memory and dependence analysis to reorder instructions, we performed instruction fusion tailored for RISC-V architectures, same as the Convolve accelerators, yielding performance benefits of more than 15% for neural networks and transformers.
In MLIR, inspired by the ZigZag schedules, we built a cost model for tiling, tailored for the Snitch architecture (Convolve-ETH) and using the Quidditch backend (Convolve-CAM). This cost model is able to select among the top 5% better performing tile sizes.
All in all, we built a modular compilation chain that is able to speed-up the Convolve use-cases and other applications, targeting RISC-V, which is at the core of the Convole accelerators.

7 References

[1] Bondhugula, U., Hartono, A., Ramanujam, J. and Sadayappan, P. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson AZ USA, June 2008), 101–113.

[2] Chollet, F. 2017. Xception: Deep Learning with Depthwise Separable Convolutions. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Honolulu, HI, July 2017), 1800–1807.

[3] Dong, C., Loy, C.C. and Tang, X. 2016. Accelerating the Super-Resolution Convolutional Neural Network. arXiv.

[4] Goetschalckx, K., Wu, F. and Verhelst, M. 2023. DepFiN: A 12-nm Depth-First, High-Resolution CNN Processor for IO-Efficient Inference. *IEEE Journal of Solid-State Circuits*. 58, 5 (May 2023), 1425–1435. https://doi.org/10.1109/JSSC.2022.3210591.

[5] He, K., Zhang, X., Ren, S. and Sun, J. 2016. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Las Vegas, NV, USA, June 2016), 770–778.

[6] High-Utilization, High-Flexibility Depth-First CNN Coprocessor For Image Pixel Processing On FPGA | PDF: *https://www.scribd.com/document/663440440/High-Utilization-High-Flexibility-Depth-First-CNN-Coprocessor-for-Image-Pixel-Processing-on-FPGA*. Accessed: 2025-09-14.

[7] Hsieh, S.-W. et al. 2025. 23.5 MAE: A 3nm 0.168mm^2 576MAC Mini AutoEncoder with Line-Based Depth-First Scheduling for Generative AI in Vision on Edge Devices. *2025 IEEE International Solid-State Circuits Conference (ISSCC)* (San Francisco, CA, USA, Feb. 2025), 414–416.

[8] https://mlir.llvm.org/docs/Passes/#-affine-loop-tile Passes: affine-loop-tile.

[9] iree/compiler/src/iree/compiler/Codegen/LLVMCPU/LLVMCPUTile.cpp at 2967df6133cb7babce591287c30ce9e2bf478d4b · iree-org/iree: *https://github.com/iree-org/iree/blob/2967df6133cb7babce591287c30ce9e2bf478d4b/compiler/src/iree/compiler/Codegen/LLVMCPU/LLVMCPUTile.cpp*. Accessed: 2025-09-14.

[10] iree/compiler/src/iree/compiler/Codegen/LLVMGPU/LLVMGPUTileAndDistribute.cpp at 2967df6133cb7babce591287c30ce9e2bf478d4b · iree-org/iree: *https://github.com/iree-org/iree/blob/2967df6133cb7babce591287c30ce9e2bf478d4b/compiler/src/iree/compiler/Codegen/LLVMGPU/LLVMGPUTileAndDistribute.cpp*. Accessed: 2025-09-14.

[11] KULeuven-MICAS/snax-mlir: Driving Snax with MLIR: *https://github.com/kuleuven-micas/snax-mlir*. Accessed: 2025-09-14.

[12] Lamprakos, C. 2025. cappadokes/idealloc.

[13] Lee, E.A. and Messerschmitt, D.G. 1987. Synchronous data flow. *Proceedings of the IEEE*. 75, 9 (1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876.

[14] Liu, D., Ji, S., Lu, K. and He, Q. 2024. Improving {Indirect-Call} Analysis in {LLVM} with Type and {Data-Flow}{Co-Analysis}. *33rd USENIX Security Symposium (USENIX Security 24)* (2024), 5895–5912.

[15] Loop Optimizations Where Blocks are Required: *https://www.intel.com/content/www/us/en/developer/articles/technical/loop-optimizations-where-blocks-are-required.html*. Accessed: 2025-09-14.

[16] Mei, L., Goetschalckx, K., Symons, A. and Verhelst, M. 2023. DeFiNES: Enabling Fast Exploration of the Depth-first Scheduling Space for DNN Accelerators through Analytical Modeling. *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Montreal, QC, Canada, Feb. 2023), 570–583.

[17] Mei, L., Houshmand, P., Jain, V., Giraldo, S. and Verhelst, M. 2021. ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators. *IEEE Transactions on Computers*. 70, 8 (Aug. 2021), 1160–1174. https://doi.org/10.1109/TC.2021.3059962.
[18] [mlir][linalg] Is tiling of linalg.fill implemented now? - MLIR: 2024. *https://discourse.llvm.org/t/mlir-linalg-is-tiling-of-linalg-fill-implemented-now/80963/5*. Accessed: 2025-09-14.
[19] Porpodas, V., Rocha, R.C.O., Brevnov, E., Góes, L.F.W. and Mattson, T. 2019. Super-Node SLP: optimized vectorization for code sequences containing operators and their inverse elements. *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (2019), 206–216.
[20] Porpodas, V., Rocha, R.C.O. and Góes, L.F.W. 2018. Look-ahead SLP: auto-vectorization in the presence of commutative operations. *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (New York, NY, USA, 2018), 163–174.
[21] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.-C. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (Salt Lake City, UT, June 2018), 4510–4520.
[22] Singh, S., Perais, A., Jimborean, A. and Ros, A. 2022. Exploring Instruction Fusion Opportunities in General Purpose Processors. *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2022), 199–212.
[23] SVF github: 2025. *https://github.com/SVF-tools/SVF*.
[24] Symons, A., Mei, L., Colleman, S., Houshmand, P., Karl, S. and Verhelst, M. 2023. Stream: A Modeling Framework for Fine-grained Layer Fusion on Multi-core DNN Accelerators. *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (Raleigh, NC, USA, Apr. 2023), 355–357.
[25] Tuning - IREE: *https://iree.dev/reference/tuning/*. Accessed: 2025-09-14.
[26] Tuning - IREE: *https://iree.dev/reference/tuning/#what-is-a-dispatch*. Accessed: 2025-09-14.
[27] tvm-rfcs/rfcs/0037-arm-ethosu-cascading-scheduler.md at main · apache/tvm-rfcs: *https://github.com/apache/tvm-rfcs/blob/main/rfcs/0037-arm-ethosu-cascading-scheduler.md*. Accessed: 2025-09-14.
[28] Xing, Y., Liang, S., Sui, L., Jia, X., Qiu, J., Liu, X., Wang, Y., Wang, Y. and Shan, Y. 2019. DNNVM : End-to-End Compiler Leveraging Heterogeneous Optimizations on FPGA-based CNN Accelerators. arXiv.
[29] 2025. *2025 EuroLLVM - An Introduction to Tensor Tiling in MLIR*.
[30] 2025. opencompl/Quidditch. opencompl.