TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Chair of Compiler Construction

Bachelor Thesis

An MLIR-based compiler flow for memristive-crossbar accelerators

Felix Reißmann

Born on: 8th May 2003 in Greiz
Matriculation number: 5003190

4th November 2024

First referee
Prof. Dr.-Ing. Jeronimo Castrillon

Second referee
Prof. Dr.-Ing. Diana Göhringer

Supervisor
Dr.-Ing. Asif Ali Khan

Statement of authorship

I hereby certify that I have authored this document entitled *An MLIR-based compiler flow for memristive-crossbar accelerators* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 4th November 2024

F. Reißmann

Felix Reißmann

TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Chair of Compiler Construction

Abstract

Compute in memory accelerators promise excellent performance and energy efficiency compared to traditional von Neumann architectures when it comes to data intensive applications. They aim to reduce the data movement bottleneck by performing computations directly on the data where it is stored. While offloading computations to CIM devices comes with time and energy savings, writing programs which take advantage of such accelerators is a time intensive, error-prone and manual process that requires a sufficient understanding of the underlying hardware.

This thesis presents a compilation flow using the Cinnamon compiler, which leverages the MLIR compiler infrastructure, to automatically recognize and offload operations which could take advantage of a memristive-crossbar accelerator. It is implemented in the Cinnamon project, which is open source and available on GitHub. The compilation flow is explained step by step, from a high-level PyTorch program down to the generated API calls to an accelerator runtime library. Finally a performance and usability evaluation is conducted.

Zusammenfassung

Compute in Memory Beschleuniger versprechen eine hervorragende Leistung und Energieeffizienz für datenintensive Anwendungen im Vergleich zu traditionellen von-Neumann-Architekturen. Sie zielen darauf ab, den Datenbewegungsengpass zu reduzieren, indem Berechnungen direkt am Speicherort der Daten durchgeführt werden. Während das Auslagern von Berechnungen auf CIM-Geräte mit Zeit- und Energieeinsparungen einhergeht, ist das Schreiben von Programmen, die von solchen Beschleunigern profitieren, ein zeitaufwändiger, fehleranfälliger und manueller Prozess, der ein umfassendes Verständnis der zugrundeliegenden Hardware erfordert.

Diese Arbeit präsentiert einen Kompilationsfluss unter Verwendung des Cinnamon-Compilers, der die MLIR-Compilerinfrastruktur nutzt, um automatisch Operationen zu erkennen und auszulagern, die von einem memristiven Crossbar-Beschleunigern profitieren. Er ist im Cinnamon-Projekt implementiert, welches Open Source auf GitHub verfügbar ist. Der Kompilationsfluss wird Schritt für Schritt von einem PyTorch-Programm bis zu den generierten API-Aufrufen an eine Beschleuniger-Laufzeitbibliothek erläutert. Abschließend wird eine Leistungs- und Benutzerfreundlichkeitsbewertung des durchgeführt.

Acronyms

CIM	Compute In Memory
CNM	Compute Near Memory
CAM	Content Addressable Memory
MLIR	Multi-Level Intermediate Representation
LLVM	Low Level Virtual Machine
API	Application Programming Interface
JIT	Just In Time
IR	Intermediate Representation
ASAP	As Soon As Possible
xBar	Crossbar
DAC	Digital to Analog Converter
ADC	Analog to Digital Converter
S&A	Shift and Accumulate
S&H	Sample and Hold

Contents

1 Introduction

This chapter aims to provide an overview of the full compilation flow structure, as well as its goals. Each of the following chapters will focus on a specific section of the conversion process.

1.1 Motivation and Goals

The compilation flow presented in this thesis aims to simplify the process of creating applications which utilize CIM or Compute Near Memory (CNM) devices. Specifically, the goal is to provide an easy-to-use interface for compiling PyTorch models in a way that offloads suitable computations to a memristive-crossbar accelerator. This would significantly reduce the barrier to entry for developers who want to utilize these devices in their applications. Additionally, models compiled with this compilation flow should benefit from the energy efficiency and high throughput of memristive-crossbar accelerators. The flow should be modular, in order to allow for easy extension and modification in the future. Extensibility is especially important, since the field of memristive-crossbar accelerators is still rapidly evolving and new devices with different characteristics are constantly being developed.

1.2 Design Paradigms

The MLIR compiler infrastructure is quickly becoming the de-facto standard for building compilers targeting novel hardware architectures. This is due to its flexibility, extensibility and a large set of builtin functionality. The presented compilation flow also builds on top of this infrastructure, as it removes the need of designing a custom compiler from scratch. MLIR provides all its functionality around a set of dialects, which are used to represent programs in different aspects as well as levels of abstraction. All dialects build upon a common syntax, making it possible to use multiple dialects in a single program. This enables the reuse of existing dialects in newly developed ones. For transformation within a dialect or conversion between dialects, MLIR provides a pass system. Passes usually focus on accomplishing a single task, making them easy to understand and less error prone. The modularity in the dialects and passes, as well as the large library of builtins, makes MLIR a powerful tool for building compilers. Third party projects like Torch-MLIR, which is also used in this compilation flow, provide further dialects and passes.

1.3 Compilation Flow Overview

As the flow focuses on the compilation of PyTorch models, a frontend is needed to convert the PyTorch model into an MLIR representation which is used during the majority of the compilation process. This frontend is provided by the Torch-MLIR project and will be discussed in detail in Chapter 2. All main conversions necessary for offloading computations to CIM devices are provided by the Cinnamon compiler, for which additional dialects and passes were developed as part of this thesis. The Cinnamon compiler aims to provide the infrastructure needed, not just for targeting CIM devices, but also other novel hardware architectures like CNM and Content Addressable Memory (CAM) devices. The conversions of the Cinnamon compiler are explored in Chapter 3. The final steps of the compilation, which need to produce executable binaries, are handled by the Low Level Virtual Machine (LLVM) project, which the MLIR project is part of. The LLVM based conversions are covered in Chapter 4. Figure 1.1 shows a simplified overview of the complete compilation flow.

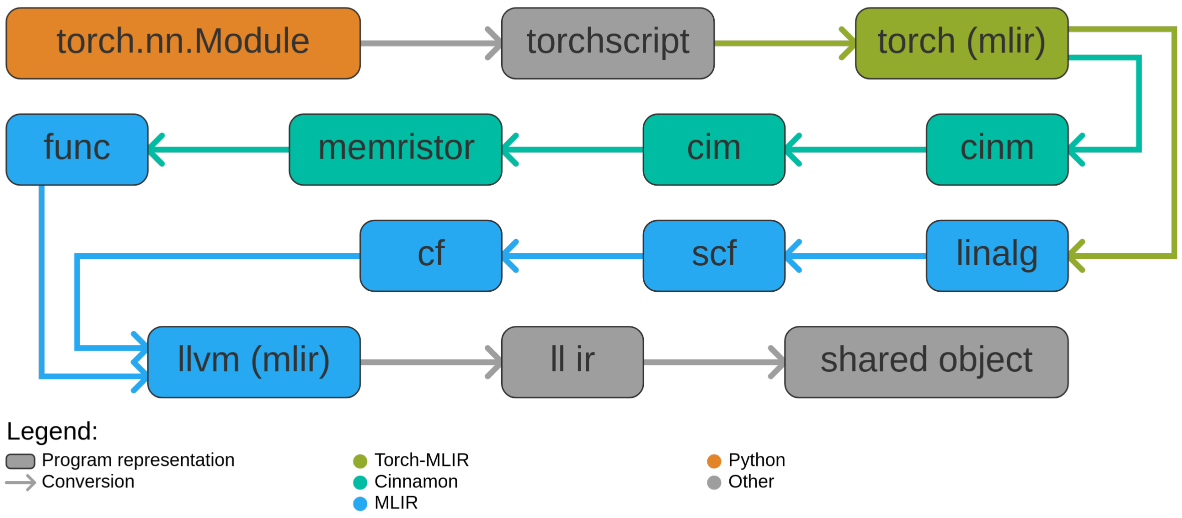

Figure 1.1: Compilation flow overview

Each row of conversions in Figure 1.1 represents a different stage of the compilation flow. The PyTorch model, which forms the input of the flow, is first compiled into `torchscript` using the PyTorch Just In Time (JIT) compilation. This `torchscript` representation of the model is then converted to the `torch` MLIR dialect. The `torch` dialect as well as the conversion from `torchscript` are provided by the Torch-MLIR project. After these steps, the model is in a form that can be used by tools built with the MLIR infrastructure. After the `torch` dialect, the compilation flow splits into two paths. The upper path is used for operations that can be offloaded to CIM devices. For this, the `torch` dialect representation is lowered by Cinnamon to the `cinm` dialect. This dialect is the main entrypoint dialect of Cinnamon and is used for representing operations that can be further lowered onto CIM or CNM devices. All operations which are not representable in the `cinm` dialect are lowered to the `linalg` dialect by passes provided by Torch-MLIR. The `linalg` representation is then lowered via the `scf` and `cf` dialects to the `llvm` dialect by builtin MLIR passes.

All operations that were previously successfully converted to the `cinm` dialect are now lowered via the `cim` dialect to the `memristor` dialect. During these conversions, further transformations are applied to reshape the operations into a form that can be executed on CIM devices. The `memristor` dialect is the target dialect for memristive-crossbar accelerators in Cinnamon and closely resembles API endpoints of runtime libraries for such CIM devices. As a final step in the Cinnamon based pipeline, all operations in the `memristor` dialect are converted to runtime API calls represented in the `func` dialect. The `func` dialect is a builtin MLIR dialect and can easily be lowered to the `llvm` dialect. At this point the split paths are rejoined as all operations are now represented in the `llvm` dialect.

The `llvm` dialect representation is then converted into LLVM Intermediate Representation (IR) and compiled into a shared object by `mlir-translate` and `clang` respectively, which are tools provided by the LLVM project. This is the final output of the compilation flow and can be executed on a host machine. The shared object contains the compiled model as well as all necessary runtime library calls to the CIM device. The runtime library is not linked into the shared object directly, as it is not part of the compilation flow itself, and needs to be provided separately.

2 Frontend

In this chapter, the conversion of PyTorch models to `torchscript` and further to the `torch` MLIR dialect will be discussed. These conversions are also shown in the top row of Figure 1.1.

The needed functionality is fully provided by the Torch-MLIR project. It can be invoked using the `torch_mlir` Python module, as shown in Listing 2.1. The same `torch_mlir` API is also wrapped in the `cinnamon` module as part of the Cinnamon PyTorch backend, which is presented in Chapter 5. For completeness, a subset of steps happening under the hood in the `torch_mlir` module are discussed in the following sections.

```python
import torch
import torch_mlir

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()

        self.fc1 = torch.nn.Linear(5, 5)
        self.fc2 = torch.nn.Linear(5, 10)
        self.fc3 = torch.nn.Linear(10, 2)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x

model = Model()
sample_input = torch.randn(5)

torch_mlir_representation = torch_mlir.compile(model, sample_input)
```

Listing 2.1: Conversion of a PyTorch model to the `torch` dialect using the `torch_mlir` Python module

2.1 Converting PyTorch Models to Torchscript

The MLIR based tools provided by Torch-MLIR need an MLIR representation of the PyTorch model and as such cannot receive Python objects directly. Instead, the model is first converted to another intermediate representation named `torchscript` . It is part of PyTorch and used mainly used for model serialization, optimization and interoperability with other tools and languages. As such, the compilation of PyTorch models to `torchscript` is provided by PyTorch itself and is easily invoked as shown on line 5 in Listing 2.2. The result of this operation is a `torch.ScriptModule` Python object, which is a wrapper around the generated `torchscript` representation of the model.

```python
1 import torch
2
3 model = Model() #torch.nn.Module, as seen in previous listing
4
5 script_module = torch.jit.script(model)
```

Listing 2.2: Conversion of a PyTorch model to `torchscript`

2.2 Converting Torchscript to the Torch Dialect

After the `torch.ScriptModule` has been created, the models `forward` function, which is the function needed for inference, is traced. The tracing step is also built into PyTorch and returns a graph of functions that are invoked when running the `forward` function. This graph is then passed to a `ModuleBuilder`, which is part of Torch-MLIR and written in C++. It converts the graph into a raw MLIR representation, where each of the traced functions is defined separately. Since the module is now encoded in MLIR, all further conversions can be done using the MLIR infrastructure. As a final step, the `forward` function in the produced MLIR code is completely flattened by inlining all subfunctions, which results in a MLIR module with a single `forward` function defined.

2.3 Additional Conversions in Torch-MLIR

The Torch-MLIR project provides a range of further conversions which use the `torch` dialect as a starting point. The conversion from the `torch` dialect to the `linalg` dialect is used by this compilation flow to lower operations that are not representable in the `cinm` dialect. As the `linalg` dialect is a MLIR builtin dialect, any further lowering towards the `llvm` dialect can be achieved using builtin MLIR passes. Torch-MLIR also provides conversions to the `tosa` dialect. TOSA stands for Tensor Operator Set Abstraction and is a dialect developed by the MLIR community to represent `tensor` operations. This dialect is currently unused by the compilation flow, but conversions from it to `cinm` might be implemented in the future to allow for a bigger set of fontends to be used with the Cinnamon compiler. Additional conversion passes are present, which can be used to convert a subset of `torch` operations to the `std` or `scf` dialects.

Torch-MLIR also provides a large set of transformation passes. Some of them are used by the compilation flow to canonicalize the MLIR code before further lowering to cinm. Others may be used to optimize, by inlining, fusing and decomposing operations.

2.4 Working with Quantized Models

The conversion of PyTorch models to the `torch` dialect works for a wide range of PyTorch models. However, there are some limitations. Memristive-crossbar devices are only able to execute matrix multiplications on integer values. Because of this, models trained with floating point precision need to be converted to use integer operations. This process is called quantization and is also provided by PyTorch. There are different strategies for quantization, but they all aim to reduce the value precision used by the model while keeping its prediction accuracy as high as possible.

The tools used in the compilation flow are currently set up using the latest LLVM (and MLIR) release 19.1.3. The Torch-MLIR version, which is compatible with this release unfortunately still has an extremely limited support for quantized models. The support has steadily improved in the latest releases of Torch-MLIR. Additionally, newer versions of LLVM with a more extensive `quant` dialect promise to provide a more complete support. With the current setup, automatically quantized models are unusable in the compilation flow. However, it is possible to manually quantize very small models which can be compiled successfully.

Manually quantizing a model consists of creating a structural clone of the original model. Each layer is recreated manually by choosing and inserting the needed quantization operations before the actual layer computation. The quantization is then undone on the result tensor. This process is very error prone and time consuming, but it is sufficient for small models as a proof of concept. Future versions of Torch-MLIR will hopefully allow for a more automated quantization process and compilation of larger models.

3 Cinnamon

This chapter will focus on all conversions performed by the Cinnamon compiler. It provides a step by step examination of all intermediate dialects and performed transformation and lowering passes. This part of the compilation flow is represented by the second row of conversions in Figure 1.1.

3.1 Conversion from Torch to Cinm Dialect

The `cinm` dialect is the main entrypoint dialect of Cinnamon. It provides a wide range of operations which may be available on specific CIM and CNM devices. The subset of `cinm` operations used by the compilation flow is shown in Table 3.1.

Operation signature	Description
`cinm.compute -> %result`	Scoped block operation containing potentially multiple `cinm.ops`
`cinm.yield %result`	Terminator operation of `cinm.compute`
`cinm.op.gemm %lhs %rhs -> %result`	General matrix-matrix multiplication
`cinm.op.gemv %lhs %rhs -> %result`	General matrix-vector multiplication

Table 3.1: Relevant `cinm` operations

Cinnamon provides a conversion pass from the `torch` dialect to the `cinm` dialect. This pass is responsible for lowering all `torch` operations that are planned to be offloaded to `cinm`. An example `torch` dialect input is shown in Listing 3.1. Currently only `torch` operations representing matrix-matrix and matrix-vector multiplications are considered for lowering. When the compilation flow is extended to support targets other than memristive-crossbar devices, additional operations may be considered as well.

For this input, the conversion pass will lower only the `torch.aten.mm` operations on lines 11 and 15 in Listing 3.1. All other operations will be left unchanged. For each lowered operation, a new `cinm.compute` block is created. This block contains the `cinm` equivalent for the replaced `torch` operation. It is terminated by `cinm.yield`, which can be thought of as returning the result to the outside region. An example of the generated operations to replace the first `torch.aten.mm` on line 11 in Listing 3.1 is shown in Listing 3.2.

```
module attributes {torch.debug_module_name = "Model"} {
  func.func @forward(%arg0: !torch.vtensor<[1,5],!torch.qint16>) ->
      ↪ !torch.vtensor<[1,2],!torch.qint16> {
    // constants and model parameters defined above:
    // %int0 : !torch.int; %int1 : !torch.int
    // %b0 : !torch.vtensor<[3],!torch.qint16>
    // %w0 : !torch.vtensor<[3,5],!torch.qint16>
    // %b1 : !torch.vtensor<[2],!torch.qint16>
    // %w1 : !torch.vtensor<[2,3],!torch.qint16>

    %tw0 = torch.aten.transpose.int %w0, %int0, %int1 : ... ->
        ↪ !torch.vtensor<[5,3],!torch.qint16>
    %m0 = torch.aten.mm %arg0, %tw0 : ... -> !torch.vtensor<[1,3],!torch.qint16>
    %r0 = torch.aten.add.Tensor %m0, %b0, %int1 : ... ->
        ↪ !torch.vtensor<[1,3],!torch.qint16>

    %tw1 = torch.aten.transpose.int %w1, %int0, %int1 : ... ->
        ↪ !torch.vtensor<[3,2],!torch.qint16>
    %m1 = torch.aten.mm %r0, %tw1 : ... -> !torch.vtensor<[1,2],!torch.qint16>
    %r1 = torch.aten.add.Tensor %m1, %b1, %int1 : ... ->
        ↪ !torch.vtensor<[1,2],!torch.qint16>
    return %r1 : !torch.vtensor<[1,2],!torch.qint16>
  }
}
```

Listing 3.1: Two layer dense model represented in the `torch` dialect

```
1 %barg0 = torch_c.to_builtin_tensor %arg0 : ... -> tensor<1x5xi16>
2 %btw0 = torch_c.to_builtin_tensor %tw0 : ... -> tensor<5x3xi16>
3 %bm0 = cinm.compute -> tensor<1x3xi16> {
4   %br0 = cinm.op.gemm %barg0, %btw0 : (...) -> tensor<1x3xi16>
5   cinm.yield %br0  : tensor<1x3xi16>
6 }
7 %m0 = torch_c.from_builtin_tensor %bm0 : ... ->
      ↪ !torch.vtensor<[1,3],!torch.qint16>
```

Listing 3.2: Replacement operations for `torch.aten.mm`

The `torch` dialect uses the value tensor type `!torch.vtensor` for all its operations. It is equivalent to the MLIR builtin `tensor` type but uses value semantics instead of reference semantics. This simplifies the analysis of operations during the conversion process from `torchscript` to the `torch` dialect, which was discussed in Section 2.1. Cinnamon uses the builtin `tensor` type for all its operations. This means that all `tensors` used by `cinm` operations need to be cast to the builtin `tensor` type. Similarly, all `tensors` generated by `cinm` operations need to be cast back to the `!torch.vtensor` type to remain compatible. The inserted cast operations can be seen in Listing 3.2 on lines 1, 2 and 7. The placement of the cast operations in the MLIR code is important. They always need to be inserted at the point where the argument to the respective cast operation was produced. If the placement is not correct, subsequent passes will fail to remove redundant casts between the types, which will cause further lowering down the line to fail.

3.2 Tiling

Once all offloadable operations have been lowered to the `cinm` dialect, a tiling pass needs to be performed. Similarly to other CIM and CNM devices, memristive-crossbar devices have a limited size of computational units. This means that the matrix sizes of the operands need to be limited to fit onto the device crossbars. The tiling pass is responsible for splitting up any matrix multiplication into multiple smaller multiplications if necessary. For this purpose, the tiling pass receives potentially tiered sizing information for the specific device that is targeted by the operation. This is needed as the pass is written in a way that it can be used for all device types supported by Cinnamon, not just devices with a single tier of computational units. The multiple sub-results produced by the split matrix multiplication are combined afterwards to recreate the original result.

This means that after the tiling pass has been applied, the original operation may now be placed in a loop, where on each iteration one sub-operation will be executed. Depending on how many crossbars are available on the target device, these operations may be able to executed in parallel. In order for this to be possible, the loop needs to be unrolled and the resulting multiple compute blocks need to be fused into a single compute block. This has currently not been implemented in Cinnamon, but should be considered in the future to allow for optimal use of all device resources. The downstream `cim` and `memristor` dialects, as well as associated passes, are already set up in a way to allow multiple concurrent operations to be dispatched and awaited.

Once the tiling pass has been successfully applied, it is guaranteed that all operations can be executed on the target device, and further lowering may proceed.

3.3 Conversion from Cinm to Cim Dialect

For CIM device targets, the next pass lowers the `cinm` ttcinm dialec`cim` the `cim` dialect. The `cim` dialect is the main dialect for all CIM devices, and as such only provides the subset of operations that can be executed on them. Additionally it introduces explicit resource acquisition and release. For each `cim` operation, the to be used device and crossbar have to be specified. The dialect also changes the semantics of each operation to be asynchronous. This means that the operations themselves only represent computation dispatch, while the `cim.barrier` operation is used to synchronize and await the computation results. This asynchronicity is needed to model multiple concurrent computations on the device. The `cim.barrier` operation also allows for different scheduling strategies to be enforced on the computation graph, allowing optimal use of all device resources. This is further explored in Section 3.4. A full list of all `cim` operations is shown in Table 3.2.

During the conversion, the nesting of operations into a compute block is removed. The scope of the computation is now defined implicitly by the `cim.acquire_device` and `cim.release_device` operations. This allows for further flexibility when inserting other computations, unrelated to those on the CIM device, between the `cim` operation dispatches and `cim.barrier` operations. This may be beneficial for performance as it allows for better utilization of the host CPU while the CIM device is busy. The result of lowering the operations in Listing 3.2 to the `cim` dialect is shown in Listing 3.3.

Operation signature	Description
`cim.acquire_device -> %deviceId`	Acquire a device
`cim.acquire_crossbar %deviceId -> %xbarId`	Acquire a crossbar on the specified device
`cim.op.gemm %xbarId %lhs %rhs -> %future`	General matrix-matrix multiplication
`cim.op.gemv %xbarId %lhs %rhs -> %future`	General matrix-vector multiplication
`cim.barrier %future`	Await results of computation associated with the future
`cim.release_crossbar %xbarId`	Release the specified crossbar
`cim.release_device %deviceId`	Release the specified device

Table 3.2: Relevant `cim` operations

```
%barg0 = torch_c.to_builtin_tensor %arg0 : ... -> tensor<1x5xi16>
%btw0 = torch_c.to_builtin_tensor %tw0 : ... -> tensor<5x3xi16>
%dev = cim.acquire_device -> !cim.deviceId
%xbar = cim.acquire_crossbar %dev : ... -> !cim.crossbarId
%fut0 = cim.op.gemm %xbar, %barg0, %btw0 : ... -> !cim.future<1x3xi16>
%bm0 = cim.barrier %fut0 : !cim.future<1x3xi16> -> tensor<1x3xi16>
cim.release_crossbar %xbar : !cim.crossbarId
cim.release_device %dev : !cim.deviceId
%m0 = torch_c.from_builtin_tensor %bm0 : ... ->
    ↪ !torch.vtensor<[1,3],!torch.qint16>
```

Listing 3.3: Replacement operations for `cinm.op.gemm` in the `cim` dialect

3.4 Scheduling of Cim Dialect Operations

As the `cim` dialect introduced explicit resource handling, all operations need to be scheduled onto the available computational resources. In the case of memristive-crossbar devices, the information regarding available crossbars was attached to the `cinm.compute` operation during the tiling pass. This information was then reattached to the generated `cim.acquire_device` operation during the lowering to the `cim` dialect.

Scheduling of `cim` operations can be performed by a range of scheduling passes, which implement different scheduling strategies. These passes built on top of a modular scheduling framework for side-effect free operations provided by Cinnamon. The scheduling framework allows for easy implementation of new scheduling strategies based on operation dependency graphs and available computational resources. The resulting modularity allows for use case specific scheduling problems to reuse existing scheduling strategies.

When implementing a scheduling pass using the scheduling framework, a set of hooks for the scheduling driver need to be specified. They are listed in Table 3.3.

The hooks are designed to be as generic as possible, making the scheduling framework usable for a wide range of operation scheduling problems. The `rescheduleOperationFilter`, `operationScheduler` and `barrierInserter` hooks are implemented specifically for the `cim` scheduling passes. They are trivial to implement, often only requiring one line of code. The `schedulingStrategy` hook is the most important hook and is responsible for generating the operation schedule. Any of the reusable scheduling strategies can be used as a functor

Hook name	Description
`rescheduleOperationFilter`	A predicate for filtering which operations will be considered for rescheduling. For `cim` scheduling passes this includes all `cim.op` operations.
`schedulingStrategy`	A chosen scheduling strategy in form of a functor which accepts a dependency graph and number of computation resources and returns an operation scheduling. The operation scheduling describes when each operation should be dispatched and awaited.
`operationScheduler`	A function which when supplied with an operation and a resource reschedules the operation onto the resource. Any changes to the position of the operation in the program will be handled by the scheduling driver and do not need to be implemented.
`barrierInserter`	A function which returns an awaiting operation linked to the passed operation. The returned operation will be inserted at the correct position by the scheduling driver.

Table 3.3: Scheduling framework hooks

for this hook. The scheduling strategies are implemented as separate classes and can be easily extended or replaced.

To invoke the scheduling driver, two parameters in addition to the hooks need to be specified. The first parameter is a list of scheduling roots, from which the operation dependency graph will be generated. The second parameter is a list of available computational resources. For the `cim` scheduling passes, both of these parameters are generated by another shared function. It analyzes the implicit computation block marked by the `cim.acquire_device` and `cim.release_device` operations and finds all values generated by `cim` tttcim operations which are escaping the block. These values represent the scheduling roots. It also inserts additional `cim.acquire_crossbar` operations for each crossbar available on the device. The results of these operations are then passed as the available computational resources to the scheduling driver. During these preparations, all `cim.barrier` operations are removed from the program in preparation for rescheduling. The collected information is forwarded to the scheduling driver, which will handle the scheduling of all `cim` operations. Listing 3.4 shows the simple and modular implementation of a `cim` scheduling pass.

3.5 Conversion from Cim to Memristor Dialect

The `memristor` dialect is the target dialect for all memristive-crossbar devices as it models common runtime API calls. A list of all `memristor` operations is shown in Table 3.4.

The lowering pass from the `cim` tttcim tttcim to `memristor` dialect converts all `cim` operations to their `memristor` counterparts. The `cim.acquire_device` and `cim.acquire_crossbar` operations are replaced by integer values representing the device and crossbar id. Additionally, operations to bufferize all `tensor` operands into `memrefs` are inserted. This is in preparation for the final conversion to the `func` dialect, which will generate runtime API

```
// Scheduler used for this pass
using Scheduler = cinm::utils::scheduling::AsapScheduler<Value>;

// Remove all barriers, find roots, get available crossbars
auto [crossbars, roots] = prepareForScheduling(acquireDeviceOp, rewriter);
Scheduler<Value> scheduler{crossbars};

// Define modular scheduling hooks
cinm::utils::scheduling::SchedulingHooks<Value> hooks{
    .rescheduleOperationFilter = isCimOp,
    .schedulingStrategy = scheduler, // simplified
    .operationScheduler = scheduleCimOpOnCrossbar,
    .barrierInserter = insertBarrierForCimOpResult};

// Run the scheduling
cinm::utils::scheduling::applyScheduling(rewriter, roots, hooks);
```

Listing 3.4: As Soon As Possible (ASAP) scheduling pass implementation for `cim` dialect

Operation signature	Description
`memristor.write_to_crossbar %xbarId %rhs`	Write operand to crossbar
`memristor.gemm %xbarId %lhs %result`	General matrix-matrix multiplication
`memristor.gevm %xbarId %lhs %result`	General vector-matrix multiplication
`memristor.barrier %xbarId`	Wait for computation to finish

Table 3.4: Relevant `memristor` operations

calls that have no representation of the `tensor` type. The generated `memref` operands are representable by data pointers and associated sizing information. The `bufferization` also requires, that the space for computation results needs to be allocated explicitly before the computation is dispatched. All allocation and `bufferization` operations are provided by the `bufferization` dialect, which is a builtin MLIR dialect. The result of lowering Listing 3.3 to the `memristor` dialect is shown in Listing 3.5.

```
%barg0 = torch_c.to_builtin_tensor %arg0 : ... -> tensor<1x5xi16>
%btw0 = torch_c.to_builtin_tensor %tw0 : ... -> tensor<5x3xi16>
%xbar = arith.constant 0 : i32
%res = bufferization.alloc_tensor() : tensor<1x3xi16>
%lhsb = bufferization.to_memref %barg0 : memref<1x5xi16>
%rhsb = bufferization.to_memref %btw0 : memref<5x3xi16>
%resb = bufferization.to_memref %res : memref<1x3xi16>
memristor.write_to_crossbar %c0_i32, %rhsb : i32, memref<5x3xi16>
memristor.gemm %xbar, %lhsb, %resb : i32, memref<1x5xi16>, memref<1x3xi16>
memristor.barrier %xbar : i32
%m0 = torch_c.from_builtin_tensor %res : ... ->
    ↪ !torch.vtensor<[1,3],!torch.qint16>
```

Listing 3.5: Example of `cim` tttcim operations lo`memristor` emristor dialect

In preparation for the final conversion to the `func` dialect, all `memristor` operations have a library call name attached to them. This name encodes the type of operation, as well as the integer type of the operands. The library call name will be used as the symbol name for the runtime API call references when converting to the `func` dialect.

3.6 Conversion from Memristor to Func Dialect

This final lowering pass converts all `memristor` operations to runtime API calls modeled in the `func` dialect, which is a builtin MLIR dialect. For each `memristor` operation, a corresponding runtime function declaration is generated. The symbol names are the library call names attached to the `memristor` operations. Parameter types are derived from the operand types of the `memristor` operations. Notably, all `memrefs` are converted to dynamically sized `memrefs`, as multiple operations of the same type, but with differently sized operands will use the same API endpoint. All function declarations needed for Listing 3.5 are shown in Listing 3.6.

```
func.func nested @memristor_write_to_crossbar_i16(i32, memref<?x?xi16>)
func.func nested @memristor_gemm_i16(i32, memref<?x?xi16>, memref<?x?xi16>)
func.func nested @memristor_barrier(i32)
```

Listing 3.6: Function declarations for `memristor` operations

3.7 Memristor Runtime Library

The lowering pass from `memristor` to `func` has created calls to runtime API functions. These functions need to be implemented in a runtime library which is then linked to the final executable. The runtime library needs to provide the actual implementation of the API calls. This can be achieved by forwarding the calls to a third party implementation or device driver. For testing and validation purposes, Cinnamon provides a simple runtime library implementation, which can execute all operations directly on the host CPU or conditionally `forward` them to a memristive-crossbar simulator written in C++. Further details regarding the simulator can be found in Chapter 6. This section will focus on the interface implementation of the runtime library in C.

After the output of the previous steps has been successfully lowered to the `llvm` dialect, as will be discussed in Chapter 4, the function definitions from Listing 3.6 will have been converted into a C compatible form as shown in Listing 3.7.

Mainly, all of the 2d `memref` parameters will have been converted to 7 parameters each:

- The base pointer to the start of the memory region referenced into by the `memref`.
- The data pointer to the actual start of the data for this `memref`.
- An integer with the offset of the first element in the data pointer.
- The number of elements in the first dimension.
- The number of elements in the second dimension.
- The stride of the first dimension.
- The stride of the second dimension.

The conversion of 1d `memrefs` used by the `memref_gevm` operation is analogous, resulting in 5 parameters.

All interface functions in the runtime library will match the signatures shown in Listing 3.7. The first step in each of the interface functions is to repackage the multiple `memref` related parameters back into a structured form. Afterwards, in the case of the runtime library included in Cinnamon, these repackaged arguments are passed to a templated C++ function

```
llvm.func @memristor_write_to_crossbar_i16(i32, !llvm.ptr, !llvm.ptr, i64, i64,
    ↪ i64, i64, i64)
llvm.func @memristor_gemm_i16(i32, !llvm.ptr, !llvm.ptr, i64, i64, i64, i64,
    ↪ i64, !llvm.ptr, !llvm.ptr, i64, i64, i64, i64, i64)
llvm.func @memristor_barrier(i32)
```

Listing 3.7: Memristor runtime library function definitions

for each operation. This removes the need for a separate implementation for each integer type.

To further avoid two implementation for matrix-matrix and matrix-vector multiplication, the `memristor_gevm` function just repackages the vector argument into a matrix. This allows the vector-matrix multiplication to be implemented in terms of the matrix-matrix multiplication. The result is then repackaged back into its vector form. Importantly this does not require any expensive copying of data, but only modification of `memref` metadata. The templated `memref_gemm` function performs the actual computation and writes the result back into the result `memref`. In both cases the `memristor_write_to_crossbar` operation is used to write the second operand into a crossbar buffer beforehand, from which it is retrieved during computation. As this runtime library implementation is currently single-threaded, the `memristor_barrier` operation is a no-op.

Before each computation, the library checks if the memristive-crossbar simulator discussed in Section 6.2 is available. If so, the computation is forwarded to the simulator instead of being executed on the host CPU. This allows for easy testing, of and with the simulator.

4 Backend

The output of the pipeline discussed in Chapter 3 is still MLIR code and as such not directly executable. This chapter will discuss the conversions shown in row 3 and 4 in Figure 1.1, which are needed to lower the MLIR code produced by Cinnamon and Torch-MLIR to LLVM IR and compile further into a shared object.

4.1 Lowering to LLVM IR

In order for any code generation using the LLVM compiler infrastructure to take place, the program has to be fully converted to the `llvm` dialect. At the current point in the compilation flow, the output of the Cinnamon pipeline includes operations from the bufferization, `memref`, `arith` and `func` dialects. Additionally, all operations that bypassed the Cinnamon pipeline are still a mix of `torch` and `func` dialects.

To allow further conversions to use builtin MLIR passes, the `torch` dialect has to be completely removed from the program. This is done by applying a set of passes which, among other things, convert all `torch` operations to the `linalg` dialect as well as replacing all `torch` types with builtin types. If these passes succeed, the `torch` dialect will no longer be present in the program. Although now no longer in the `torch` dialect, all operations which bypassed the Cinnamon pipeline are still represented at a higher level of abstraction than the operations that went through it. To bring all operations to a similar level, next, multiple bufferization and canonicalization passes are applied. These passes will remove the `tensor` type from the program and replace it with `memref` types. The argument and return types of the `forward` function will now also become`memref`s. During the canonicalization passes, the now redundant `tensor` casts introduced by the `torch` to `cinm` conversion will also be removed. Next, the `linalg` dialect is lowered, first to the `scf` dialect and then further to the `cf` dialect. This will bring all operations to a similarly low level of abstraction. The `arith`, `func` and `cf` dialects can now be converted to the `llvm` dialect. Finally, also the `memref` dialect is converted to llvm, again changing the signature of the main inference function to a format similar to the one discussed in Section 3.7. As a final preparation step for the export to LLVM IR, a canonicalization pass is applied to remove any remaining redundant operations. Additionally the `reconcile-unrealized-casts` and `llvm-legalize-for-export` passes are applied to remove any remaining unsupported operations and to legalize the program for export to LLVM IR. If all of the mentioned passes succeed, the program is now ready for export.

4.2 Code Generation

For the conversion from the `llvm` MLIR dialect to LLVM IR, the `mlir-translate` tool, provided by the LLVM project, is used. This tool is able to convert between the slightly different syntaxes of the `llvm` MLIR dialect and the LLVM IR. After invoking `mlir-translate` and generating LLVM IR, the `clang` compiler is used to compile the LLVM IR into a shared object. As shown in Figure 1.1, the shared object is the final product of the compilation flow. Running inference using this compiled PyTorch model will be discussed in Chapter 5.

5 PyTorch Backend

To make the compilation flow usable directly from Python, without manually invoking all the steps discussed in the previous chapters, the Cinnamon project also includes a Python module. This module provides, similarly to the `torch_mlir` module from Torch-MLIR, an easy-to-use, high level interface to the compilation flow. The process of compiling, loading and running inference on a PyTorch model using the `cinnamon` Python module is shown in Listing 5.1.

```python
import torch
from cinnamon.torch_backend.cinm import CinmBackend

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()

        self.fc1 = torch.nn.Linear(5, 5)
        self.fc2 = torch.nn.Linear(5, 10)
        self.fc3 = torch.nn.Linear(10, 2)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x

model = Model()
sample_input = torch.randn(5)

backend = CinmBackend()

compiled_model = backend.compile(model, sample_input)
model_invoker = backend.load(compiled_model)
```

Listing 5.1: Example of compiling, loading and running inference on a PyTorch model using the Cinnamon PyTorch backend

5.1 Compiling PyTorch Models

The `cinnamon` Python module provides a `CinmBackend` class which can be used to invoke the full compilation flow from Python. The class has a `compile` method as seen in Listing 5.1 line 21, which takes a module and a sample input tensor as arguments. Internally, the module will first be converted into a `torch` dialect MLIR representation using the `torch_mlir.compile` function discussed in Chapter 2. An analysis pass then extracts the signatures of all functions defined in the `torch` dialect MLIR code. This information is laster required for loading the compiled model back into Python.

The CinmBackend class currently defines a static pass pipeline for the lowering, which targets memristive-crossbar accelerators. This pipeline together with the MLIR representation of the PyTorch model is then passed to the Cinnamon pass runner. The pass runner applies all passes in the order specified by the pipeline. This results in the MLIR code being lowered to the `llvm` dialect as discussed in Chapter 3 and Chapter 4. Next the `mlir-translate` tool as well as `clang` are invoked to convert the `llvm` MLIR code to LLVM IR, and compile into a shared object. The shared object and the previously extracted signatures are then packaged into a `CompiledModel` Python object and returned to the user.

5.2 Loading of Compiled PyTorch Models

The `CompiledModel` object produced by the previous step provides functionality for saving and loading itself to and from disk. This may be useful in cases where the compiled model is distributed rather than the original PyTorch model. Or in cases where recompiling the model is not desired.

In order for the compiled model to become useable, it has to be loaded and made accessible from Python. This functionality is provided by the `load` method of the `CinmBackend` class as seen on line 22 in Listing 5.1. The `load` method takes the `CompiledModel` object as an argument and returns a texttttModelInvoker object. This object aims to provide a similar interface to the original PyTorch model, in order to make the transition from the PyTorch model to the compiled model as seamless as possible. The `load` method internally just forwards the `CompiledModel` object to the `ModelInvoker` constructor along with a list of paths to runtime libraries. An example of such a runtime library is the `memristor` runtime presented in Section 3.7.

The `ModelInvoker` constructor then first loads the specified runtimes into the Python environment. This is done using the dll loader provided by the standard `ctypes` module. After loading all runtimes, the shared object stored in the `CompiledModel` can also loaded. As the `ModelInvoker` should provide a similar interface to the original PyTorch model, it also needs to have the relevant functions. Using the list of extracted signatures, which is also stored in the `CompiledModel`, the `ModelInvoker` object creates wrappers for all functions defined in the shared object and registers them as methods of itself. The structure of the wrapper functions will be discussed in Section 5.3. After all functions have been wrapped, the `ModelInvoker` object is returned to the user.

5.3 Forwarding Calls to the Compiled Model

During the lowering process of the PyTorch model, the required parameters to the `forward` functions were converted from the original PyTorch tensors first to `memref` types and then to a set of pointers and integers. As such, the interface cannot directly be used with PyTorch tensors anymore. In order to hide this complexity from the user, each function in the shared object is wrapped by the `ModelInvoker` object.

The created wrapper functions need to perform two main tasks. First, they need to convert the passed arguments to the format expected by the function in the shared object. Second, they need to provide the storage for the result tensor which is passed to the function in the same way as the other arguments, but requires some additional handling after the function has been invoked. During the lowering described in Section 5.1, the `llvm-request-c-wrappers` pass was run. This pass resulted in the creation of C interface wrappers in the shared object. These wrappers take `memref` arguments as pointers to a `memref` descriptor structure instead of the set of pointers and integers. This allows for cleaner code in the Python wrapper functions.

To achieve task one, each tensor argument needs to be converted to a `memref` descriptor structure. This is done by first enforcing the tensor to be contiguous in memory and then extracting the data pointer, the sizes and the strides of the tensor. These are written to the `memref` descriptor structure expected by the requested C wrappers. The structure is created with a C compatible layout by using the `ctypes` module. Finally pointers to the created structures are passed to the function in the shared object.

The first step for the second task is to create a result tensor with the same shape and data type as the expected return type from the original PyTorch model. This information is looked up in the extracted signatures. The created result tensor is then converted to the memref descriptor structure in the same way as the input tensors. After the function has been invoked with all generated arguments, the result tensor may not have been written to directly. In some rare cases, an optimization may have caused the data pointer in the `memref` descriptor structure to be updated, instead of the data it originally pointed to. This change would not be reflected in the Python result tensor object. If this is detected, the data at the changed pointers is copied back into the result tensor. Finally, it is returned to the user.

Although the Torch-MLIR compilation currently only compiles the models `forward` function, which is needed for inference, the compilation and loading process in the `cinnamon` module is already designed to support multiple functions, should this become necessary in the future. Additionally, only argument of the tensor type are allowed during the Torch-MLIR compilation. The wrapper functions in the `ModelInvoker` object however are designed to support wrapping of any parameter type, as long as a wrapper object was registered for it. This allows for easy extension of the compilation flow to support additional types of parameters in the future.

5.4 Running Inference on Compiled PyTorch Models

The `ModelInvoker` object returned by the `load` method provides a similar interface to the original PyTorch model. This means that the user can call the `forward` method of the `ModelInvoker` object with the same arguments as the original PyTorch model. As discussed in the previous section, the `ModelInvoker` will then handle all necessary conversions and invoke the underlying function in the shared object. The result tensor is then returned to the user in the same format as the original PyTorch model would have returned it.

Other operations than inference, notably training are not supported by the `ModelInvoker` object. This is due to the fact that the compiled model does not include the necessary operations for training, such as backpropagation. Additionally, any changes to the model, such as modifying weights, are not possible. The compiled model is a static representation of the model at the time of compilation and cannot be changed afterwards without recompiling.

6 Usability Analysis and Evaluation

This chapter will evaluate the usability of the compilation flow and provide some benchmarking results based on a simulated memristive-crossbar accelerator. An evaluation on actual hardware is not possible at this time, as no memristive-crossbar accelerator is currently available. The simulator is based on a existing hardware architecture and resulting performance numbers should be indicative of what can be expected on actual hardware.

6.1 Usability Analysis

Because the compilation flow is built on top of the MLIR compiler infrastructure, it is highly modular in nature. Extending it with alternative frontends only requires adding a conversion from the frontends intermediate representation to the `cinm` MLIR dialect used as an entrypoint for the Cinnamon compiler. Conversion passes at this level mostly consist of one-to-one mappings of operations to the `cinm` dialect. In some cases the addition of new conversion passes may not even be necessary, as already supported dialects may be used as a stepping stone to the `cinm` dialect. The same is true for adding support for additional target devices. The `memristor` dialect provides a solid foundation for new backend dialects to be built on top of. For some devices, the operations may even be directly mappable to the device's runtime library calls. Writing a runtime library like discussed in Section 3.7, which forwards the calls to the device's runtime would be sufficient in this case. The Cinnamon compiler is built with this kind of extensibility in mind and offers good usability to developers who want to extend the compilation flow.

The PyTorch backend discussed in Chapter 5 provides a high level interface to the compilation flow. It abstracts away the details of the compilation and loading of the compiled model. Because the end user only needs to provide a PyTorch model and a sample input tensor for the compilation to work, the process should be easily integrable into existing projects. Once the model has been compiled, the `ModelInvoker` object provides a similar interface to the original PyTorch model. This allows for simple substitution of the original model with the compiled model. The use of the compilation and loading functionality only requires minimal changes to the existing codebase. A major roadblock for the usability of the end user is the lack of support for quantized models, as discussed in Section 2.4. This is a limitation which is expected to be resolved by future releases of the Torch-MLIR project. When this is the case, no additional changes to the rest of the compilation flow should be necessary. The quantized models will be able to be compiled in the same way as already described. Because of this, being only a temporary roadblock, the overall usability of the compilation flow for the user can still be considered high.

6.2 Simulator Architecture

In order to evaluate the performance and energy efficiency of the compiled models, a simulator implemented. It is designed to be close enough to existing hardware to provide meaningful results. The simulator is written in C++ with SystemC for the hardware simulation. An overview of the simulated architecture is shown in Figure 6.2.

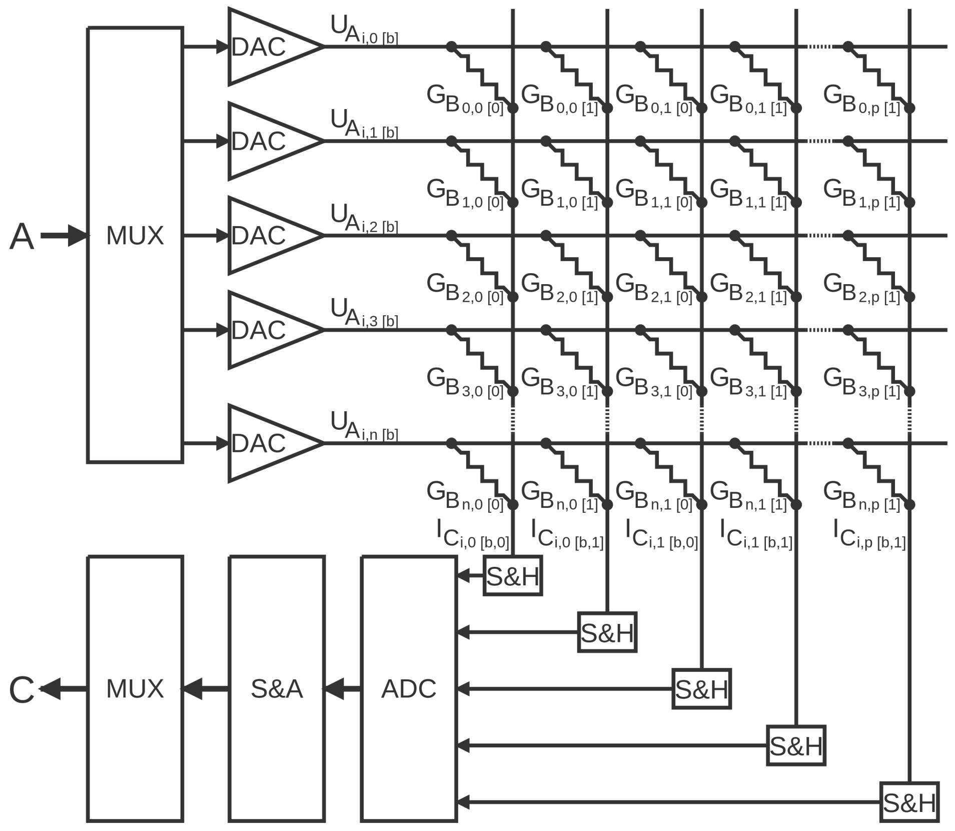
A
MUX
DAC
$U_{A_{i,0\,[b]}}$
$U_{A_{i,1\,[b]}}$
$U_{A_{i,2\,[b]}}$
$U_{A_{i,3\,[b]}}$
$U_{A_{i,n\,[b]}}$
$G_{B_{0,0\,[0]}}$ $G_{B_{0,0\,[1]}}$ $G_{B_{0,1\,[0]}}$ $G_{B_{0,1\,[1]}}$ $G_{B_{0,p\,[1]}}$
$G_{B_{1,0\,[0]}}$ $G_{B_{1,0\,[1]}}$ $G_{B_{1,1\,[0]}}$ $G_{B_{1,1\,[1]}}$ $G_{B_{1,p\,[1]}}$
$G_{B_{2,0\,[0]}}$ $G_{B_{2,0\,[1]}}$ $G_{B_{2,1\,[0]}}$ $G_{B_{2,1\,[1]}}$ $G_{B_{2,p\,[1]}}$
$G_{B_{3,0\,[0]}}$ $G_{B_{3,0\,[1]}}$ $G_{B_{3,1\,[0]}}$ $G_{B_{3,1\,[1]}}$ $G_{B_{3,p\,[1]}}$
$G_{B_{n,0\,[0]}}$ $G_{B_{n,0\,[1]}}$ $G_{B_{n,1\,[0]}}$ $G_{B_{n,1\,[1]}}$ $G_{B_{n,p\,[1]}}$
$I_{C_{i,0\,[b,0]}}$ $I_{C_{i,0\,[b,1]}}$ $I_{C_{i,1\,[b,0]}}$ $I_{C_{i,1\,[b,1]}}$ $I_{C_{i,p\,[b,1]}}$
S&H
C
MUX
S&A
ADC

Figure 6.1: Simulated memristive-crossbar accelerator architecture with mapping of $A_{m\times n} \cdot B_{n\times p} = C_{m\times p}$. Indices i and $[b]$ denote time multiplexing and bit slicing steps respectively. Used value types are assumed to fit in two bit slices.

Figure 6.1 shows the structure of a typical memristive-crossbar accelerator. In order to perform matrix multiplications, both input matrices need to be mapped onto the hardware.

In this simulator the left hand side matrix, designated with A in Figure 6.1, is mapped to the input lines entering the crossbar from the left. As these lines are laid out in one dimension, only one row of the matrix is mapped to them at a time. This results in a time multiplexed mapping of the matrix. Additionally, the values in the respective matrix rows need to be converted to analog voltage levels in order for the computation to be performed by hardware. This is achieved by a digital to analog converter. Because of hardware complexity and imperfections in analog circuitry, they cannot directly reproduce the resolution of the input values. The Digital to Analog Converter (DAC)s in the simulator are modeled to have a resolution of 2 bits. In extreme hardware configurations, the use of a DAC may be skipped completely resulting in a one bit resolution. Because of this resolution limit, an additional time multiplexing step is used where only two bits of the input value are sent at a time. The two levels of multiplexing just described are represented by the multiplexer block, positioned before the DACs in Figure 6.1. All effects of multiplexing the left hand side matrix are then reverted after the Analog to Digital Converter (ADC) conversion by the second multiplexer. This is achieved by shifting the bits back to their original position and writing the results into the correct location in the output matrix, designated with C in Figure 6.1.

The right hand side matrix, B, is mapped to the crossbar array. The simulator models a crossbar with size 128x128. Because the memristors are arranged in a 2d structure, the full matrix can be mapped at once. Similarly to the DACs for the left hand side matrix, the conductances of the memristors in the crossbar are also limited in resolution, because of differences in the physical material as well as of drifting over time. The memristors in the simulator are modeled to have a resolution of 2 bits. The bit slicing for the memristors is achieved spatially, by mapping bit slices to neighboring crossbar columns. Spatial multiplexing is used in order to keep calculation times to a minimum. The multiplexing is undone by the shift-accumulate unit after the ADC conversion. On some hardware architectures, it is possible to disable crossbar columns which remain unused in order to save energy. This is not modeled in the simulator.

The way the matrices are mapped onto the hardware allows the dot product of the respective rows and columns, which is the basic operation used during matrix multiplication, to be computed completely analog. The multiplication is performed according to Ohm's law. With U being the voltage representing a value from the left hand side matrix and G being the conductance representing a value from the right hand side, the result is calculated as $I = U \cdot G$. The addition is then performed according to Kirchhoff's current law, as the currents from all memristors in a column add up to produce the result. The result is then held steady during ADC conversion by the sample and hold units and converted back to a digital value. After all effects of multiplexing are undone, this results in a full matrix multiplication being performed.

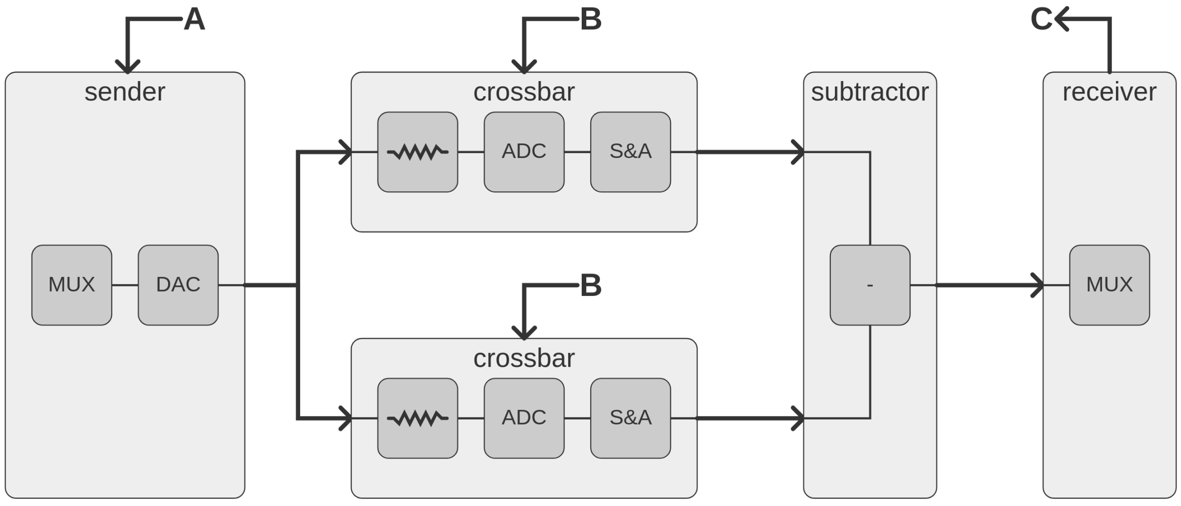

Figure 6.2: Simulator architecture

To model the architecture shown in Figure 6.1, the simulator is split into four main components, which can be seen in Figure 6.2. The first component is the sender. It is responsible for sequentially sending the bit-sliced rows of the left hand side matrix as voltages to the two crossbars which are connected. These are needed because the simulator uses differential encoding for the conductances. When using differential encoding, values from the right hand side matrix are encoded as the difference between corresponding conductances in both crossbars. For some memristor technologies this solves the problem of representing a zero conductance. It also reduces the effects of drift in the memristors, as it is expected to be similar in both crossbars and therefore cancels out during the subtraction. Additionally, differential encoding allows the representation of negative values, which is not currently used in the simulator. Both crossbars compute the dot product according to Ohm's law and Kirchhoff's current law as discussed above using their respective conductance values. They also perform the analog to digital conversion as well as the shift-accumulate operations. The results of the crossbars are then sent to the subtractor component. This component recombines the two results by subtracting them from each other, thereby decoding the differential encoding. The final result is then sent to the receiver component. It undoes the two layers of time multiplexing introduced by the sender and writes the result to the output matrix.

The simulator is set up to directly receive binary representations of the input matrices. It can dynamically accept a range of integer widths and matrix sizes, using a fully type erased representation internally. In order to simplify testing and benchmarking, an additional wrapper executable was implemented which creates a unix domain socket on the filesystem. The runtime library discussed in Section 3.7 checks for the existence of this socket and sends the input matrices to the simulator if it is found. The wrapper then executes the simulator and sends the result back to the runtime library. The simulator not only returns the output matrix to the wrapper on its invocation, but also collects and sends back performance metrics. Based on these metrics, estimates of performance and energy efficiency can be made.

6.3 Benchmarking

This section will present the results of a limited benchmarking run, aimed at demonstrating the performance and energy efficiency benefits of memristive-crossbar devices. The benchmarking was performed on a simulated accelerator described in Section 6.2. Memristor and driving parameters were chosen to similar values as are found in existing hardware. An overview of all accelerator parameters is shown in Table 6.1.

Parameter	Value
Crossbar size	128x128
DAC resolution	2 bits
Memristor resolution	2 bits
Clock frequency	1 GHz
Memristor range	10 kΩ to 40 kΩ
Maximum crossbar input voltage	0.2 V
Conductance encoding	Differential, parallelized

Table 6.1: Simulated architecture parameters

As performance analysis is not the main topic of this thesis, the benchmark consists of only a single matrix multiplication. It is sized to completely fill the simulated crossbar and is as such representative of a typical workload for previously tiled large matrix multiplications. The specific parameters of the matrix multiplication are shown in Table 6.2.

The runtime of only the computation pipeline, presented in Figure 6.1, was simulated to be 85 ns. It is comprised of 6 cycles, 1 ns each, which are needed to fill the pipeline and produce the first 2 bit slice of the result, as well as the following 79 cycles which complete the computations for the remaining 79 slices. The full computation latency is comprised of this runtime, together with the transfer latency as well as the write latency generated while

Parameter	Value
Lhs matrix size	10x128
Rhs matrix size	128x16
Integer width	16 bit
Value range	uniform random, full 16 bit range

Table 6.2: Benchmark parameters

setting up the memristor conductances. With the completely filled crossbars in this example, a total of 32768 memristor writes have to be performed. Values for the additional latencies are highly dependent on the hardware architecture and the memristor technology being used and are not provided here.

Estimation of energy consumption is equally dependent on the specific hardware architecture and memristor technology. With the assumed values for the memristor range and input voltage, the simulator estimates the energy consumption of purely the crossbar during the full computation to be about 1750 pJ. Additionally, the energy consumption of the DACs, ADCs, digital logic and memristor writes have to be taken into account. A paper presenting the OCC compiler [Sie+22] provides some estimates for the energy consumption of these components in Table II from Chapter V. Please note that the experimental setup used in the paper is different from the one used here and that shown values might not scale linearly with the crossbar size.

As a comparison, the same matrix multiplication was performed on a CPU with a single threaded loop based matrix multiplication implementation written in C++. The benchmarking environment is described in Table 6.3.

Parameter	Value
CPU	Intel(R) Core(TM) i7-8550U 4 cores, 8 threads 1.8 GHz base clock, 4.0 GHz boost clock
Compilation setup	Clang 19.1.1 with -O3 optimization flag

Table 6.3: CPU setup

The computation was performed using a warm cache and averaged a runtime of 16.24 us. This shows the large performance benefits of memristive-crossbar devices over traditional computing architectures. The energy consumption of the CPU was not measured, as this would require extensive instrumentation. However, it can be assumed to be significantly higher than the energy consumption of the accelerator.

7 Epilogue

7.1 Conclusion

This thesis presented an end-to-end compilation flow for memristive-crossbar accelerators. To make the process more accessible a Python package was developed, which provides a high-level interface to the compilation flow. The package allows for easy compilation, loading, and running of inference on compiled PyTorch models. The compilation flow was evaluated based on its usability and performance. The results show, that the compilation flow is highly modular and extensible. The PyTorch backend provides a high-level interface to the compilation flow, making it easy to integrate into existing projects. Additionally, a simulator of a existing memristive-crossbar accelerator architecture was developed to provide validation of correctness as well as evaluation of performance and energy efficiency.

7.2 Outlook

The Cinnamon compiler already supports code generation for CNM devices. Because of this, the presented compilation flow can be extended to support CNM devices as well. Future additions might also include support for CAM devices, which opens up a large set of offloading possibilities.

The PyTorch frontend is currently the only frontend supported by the compilation flow. Additional frontends supporting machine learning frameworks like ONNX and TensorFlow would further increase the usability and accessibility of the compilation flow. Import of some ONNX and TensorFlow models is already possible through conversions built into PyTorch and Torch-MLIR as well as other third-party tools.

Furthermore, the implementation of optimization passes in the various dialects of the Cinnamon compiler would be beneficial. Some optimization opportunities, like loop unrolling and compute block fusion, were already mentioned in this thesis. A range of other optimization passes are certainly possible and could improve the performance, especially of large compiled models.

Currently, the main roadblock for the compilation flow is the lack of support for automatically quantized models in Torch-MLIR. This limitation will most likely be resolved in the near future, as the support has already steadily improved during the writing of this thesis.

Acknowledgements

I would like to express my sincere gratitude to my main supervisor, Dr.-Ing. Asif Ali Khan, for his guidance and support throughout the duration of my thesis. His expertise and insightful feedback have been invaluable in shaping the direction and quality of this work. I am also deeply thankful to Clément Fournier and Hamid Farzaneh for their continuous assistance and willingness to discuss various aspects of the project. Their contributions have been crucial to the successful completion of this thesis.

I extend my heartfelt thanks to Prof. Dr.-Ing. Jeronimo Castrillon and Prof. Dr.-Ing. Diana Göhringer for kindly agreeing to serve as the referees for my thesis.

Thank you all for your contributions and support.

Bibliography

[He+20] Wangxin He et al. “2-Bit-Per-Cell RRAM-Based In-Memory Computing for Area-/Energy-Efficient Deep Learning”. In: *IEEE Solid-State Circuits Letters* 3 (2020), pp. 194–197. DOI: 10.1109/LSSC.2020.3010795.

[Jia+21] Yanhai Jiang et al. “HARNS: High-level Architectural Model of RRAM based Computing-in-memory NPU”. In: *2021 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*. 2021, pp. 35–36. DOI: 10.1109/ICTA53157.2021.9661827.

[Lat+21] Chris Lattner et al. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[Qu+21] Songyun Qu et al. “ASBP: Automatic Structured Bit-Pruning for RRAM-based NN Accelerator”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 745–750. DOI: 10.1109/DAC18074.2021.9586105.

[Yu+21] Shimeng Yu et al. “RRAM for Compute-in-Memory: From Inference to Training”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.7 (2021), pp. 2753–2765. DOI: 10.1109/TCSI.2021.3072200.

[Sie+22] Adam Siemieniuk et al. “OCC: An Automated End-to-End Machine Learning Optimizing Compiler for Computing-In-Memory”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.6 (2022), pp. 1674–1686. DOI: 10.1109/TCAD.2021.3101464.

[Diw+23] Sumit Diware et al. “Accurate and Energy-Efficient Bit-Slicing for RRAM-Based Neural Networks”. In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 7.1 (2023), pp. 164–177. DOI: 10.1109/TETCI.2022.3191397.

[FSR24] M. Fritscher, S. Singh, and T. et al. Rizzi. “A flexible and fast digital twin for RRAM systems applied for training resilient neural networks”. In: *Scientific Reports* 14 (2024), p. 23695. DOI: 10.1038/s41598-024-73439-z. URL: https://doi.org/10.1038/s41598-024-73439-z.

[Kum+24] Ashwani Kumar et al. “Energy Efficient Implementation of MVM Operations Using Filament-Free Bulk RRAM Array”. In: *2024 Neuro Inspired Computational Elements Conference (NICE)*. 2024, pp. 1–5. DOI: 10.1109/NICE61972.2024.10549369.

[Kha+25] Asif Ali Khan et al. “CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’25)*. ASPLOS ’25. Rotterdam, The Netherlands: Association for Computing Machinery, Mar. 2025.