Project:

MNEMOSENE

(Grant Agreement number 780215)

"Computation-in-memory architecture based on resistive devices"

Funding Scheme: Research and Innovation Action

Call: ICT-31-2017 "Development of new approaches to scale functional performance of information processing and storage substantially beyond the state-of-the-art technologies with a focus on ultra-low power and high performance"

Date of the latest version of ANNEX I: 11/10/2017

D2.4 – Complete Parallelization, Orchestration and Compilation Flow

Project Coordinator (PC):	Prof. Said Hamdioui Technische Universiteit Delft - Department of Quantum and Computer Engineering (TUD) Tel.: (+31) 15 27 83643 Email: S.Hamdioui@tudelft.nl
Project website address:	www.mnemosene.eu
Lead Partner for Deliverable:	INRIA
Report Issue Date:	30/07/2020

Document History
(Revisions – Amendments)

Version and date	Changes
1.0 30/07/2020	First version

Dissemination Level

PU	Public	X
PP	Restricted to other program participants (including the EC Services)	
RE	Restricted to a group specified by the consortium (including the EC Services)	
CO	Confidential, only for members of the consortium (including the EC)	

European Commission

The MNEMOSENE project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No 780215

The MNEMOSENE project aims at demonstrating a new computation-in-memory (CIM) based on resistive devices together with its required programming flow and interface. To develop the new architecture, the following scientific and technical objectives will be targeted:

- Objective 1: Develop new algorithmic solutions for targeted applications for CIM architecture.
- Objective 2: Develop and design new mapping methods integrated in a framework for efficient compilation of the new algorithms into CIM macro-level operations; each of these is mapped to a group of CIM tiles.
- Objective 3: Develop a macro-architecture based on the integration of group of CIM tiles, including the overall scheduling of the macro-level operation, data accesses, inter-tile communication, the partitioning of the crossbar, etc.
- Objective 4: Develop and demonstrate the micro-architecture level of CIM tiles and their models, including primitive logic and arithmetic operators, the mapping of such operators on the crossbar, different circuit choices and the associated design trade-offs, etc.
- Objective 5: Design a simulator (based on calibrated models of memristor devices & building blocks) and FPGA emulator for the new architecture (CIM device combined with conventional CPU) in order demonstrate its superiority. Demonstrate the concept of CIM by performing measurements on fabricated crossbar mounted on a PCB board.

A demonstrator will be produced and tested to show that the storage and processing can be integrated in the same physical location to improve energy efficiency and also to show that the proposed accelerator is able to achieve the following measurable targets (as compared with a general purpose multi-core platform) for the considered applications:

- Improve the energy-delay product by factor of 100X to 1000X
- Improve the computational efficiency (#operations / total-energy) by factor of 10X to 100X
- Improve the performance density (# operations per area) by factor of 10X to 100X

LEGAL NOTICE

Neither the European Commission nor any person acting on behalf of the Commission is responsible for the use, which might be made, of the following information.

The views expressed in this report are those of the authors and do not necessarily reflect those of the European Commission.

© MNEMOSENE Consortium 2020

Table of Contents

1 Introduction....4
2 Requirements and Changes over Deliverable 2.3....4
2 MLIR: A Multilevel Intermediate Representation....5
3 Overview of the Revised Compilation Flow....6
4 Illustration of the Compilation flow on the HAR Kernel....8
5 Building and using the Compiler....11
6 Conclusions....13
References....13

1 Introduction

The objectives of the MNEMOSENE project include the improvement of the energy-delay product, the computational efficiency and performance density by several orders of magnitude compared to state-of-the-art architectures. A cornerstone of the proposed solution is the memristor-based Compute-in-Memory (CIM) architecture, which eliminates long-distance, high-latency data transfers between memory and computing units required in conventional Von Neumann-based architectures by carrying out computations for performance-critical operations directly in memory.

In order for applications to benefit from this architecture, their operations must be divided into highly parallel, uniform operations eligible for in-memory computation and control logic that cannot benefit from CIM and that must be carried out by conventional computing devices. It is crucial for this process that as many eligible operations as possible are identified and effectively processed in memory, resulting only in as few computations as possible carried out on the conventional cores.

The programmability of the CIM architecture is a key factor for its overall success. Manual identification of eligible operations and mapping to hardware resources is tedious, error-prone and requires detailed knowledge of the target architecture and therefore does not represent a viable approach to program the CIM architecture. The goal of **Task 2.2** is to provide a compilation flow that unburdens programmers from technical details by allowing them to express algorithms at a high level of abstraction and that automates parallelization, orchestration and the mapping of operations to the CIM architecture. **Deliverable 2.4** represents the final version of the compilation infrastructure resulting from this task.

This document provides an overview of the solution for **Deliverable 2.4**.

2 Requirements and Changes over Deliverable 2.3

The requirements for the proposed compilation infrastructure are defined by the results of **Work Package 1**, **Work Package 3**, and **Work Package 5**, respectively identifying targeted applications and their requirements, defining the high-level target architecture, and developing the full-system simulator. These requirements can be summarized as follows:

- Support for an adequate high-level representation for performance-critical operations of applications identified in **Work Package 1**, abstracting from technical details of the target architecture from **Work Package 3**.
- Fully automatic identification and parallelization of operations that can efficiently be carried out by the CIM architecture, in particular matrix-matrix operations and matrix-vector operations as identified in **Work Package 1**.
- Full end-to-end compilation flow from the high-level representation to code that executes on the CIM architecture proposed in **Work Package 3**.
- Integration with the full system simulator provided by **Work Package 5**.

The proposed solution is an iteration of the solution proposed from **Deliverable 2.3** with the main modifications summarized below:

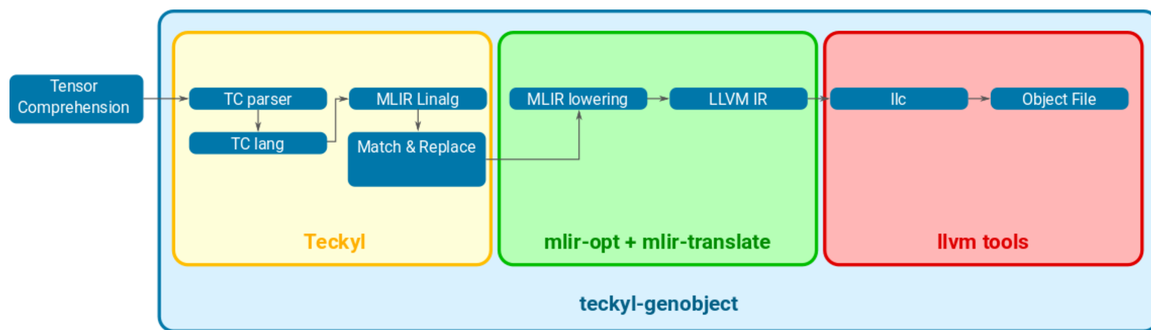

Figure 1: Compilation Flow

- The flow has been simplified and unburdened from transformations inherited from the Tensor Comprehensions framework [1] (e.g., the additional representation in the *Halide framework* [2] used the previous iteration has been dropped entirely).
- The compiler has been rebased on the extensible, multi-level intermediate representation MLIR [4], which allows for precise modeling of the high-level operations of interest (matrix-vector products and matrix-matrix products) compared to the previously used schedule trees [3], leverages existing MLIR infrastructure and allows for integration with MLIR-based tools.
- The source-to-source approach limited to polyhedral representations due to the use of the integer set library *isl* has been replaced with a code generation path from a high-level MLIR representation to low-level LLVM IR. Code generation has been aligned to standard tools from the LLVM ecosystem (e.g., *llc*) for improved interoperability.

Since the switch to MLIR represents a major change, the next section provides an overview of MLIR.

2 MLIR: A Multilevel Intermediate Representation

MLIR [4] is a multi-level intermediate representation that recently became part of the LLVM project repository. Instead of representing operations in a single, monolithic intermediate representation common in general-purpose compilers, MLIR is designed as an extensible set of *dialects*, each defining their own set of *operations*. Dialects may model operations from various levels of abstraction and can co-exist within the same module, which allows for the preservation of high-level information throughout the compilation process. Examples of MLIR dialects are:

- The *linalg* dialect, modeling linear algebra operations (e.g. matrix products, element-wise operations, etc.).
- The *scf* dialect, modeling static control flow (e.g., for loops with static control).
- The *std* dialect, providing common, low-level operations, such as integer and floating-point arithmetic, function calls and memory accesses.
- The *llvm* dialect, providing an embedding of the well-known LLVM IR into MLIR.

Figure 2 shows an example of two representations of a matrix multiplication in MLIR. The representation on the left side of the figure uses lower-level operations from the *std* and *scf* dialects, while the representation on the right side is based on the high-level *matmul* operation from the *linalg* dialect.

```
func @mm(%arg0: memref<?x?xf32>,
         %arg1: memref<?x?xf32>,
         %arg2: memref<?x?xf32>)
{
    %c0 = constant 0 : index
    %c1 = constant 1 : index
    %0 = dim %arg0, %c0 : memref<?x?xf32>
    %1 = dim %arg0, %c1 : memref<?x?xf32>
    %2 = dim %arg1, %c1 : memref<?x?xf32>
    scf.for %arg3 = %c0 to %0 step %c1 {
      scf.for %arg4 = %c0 to %2 step %c1 {
        scf.for %arg5 = %c0 to %1 step %c1 {
          %3 = load %arg1[%arg5, %arg4] : memref<?x?xf32>
          %4 = load %arg0[%arg3, %arg5] : memref<?x?xf32>
          %5 = mulf %4, %3 : f32
          %6 = load %arg2[%arg3, %arg4] : memref<?x?xf32>
          %7 = addf %5, %6 : f32
          store %7, %arg2[%arg3, %arg4] : memref<?x?xf32>
        }
      }
    }
    return
}
```

```
func @mm(%arg0: memref<?x?xf32>,
         %arg1: memref<?x?xf32>,
         %arg2: memref<?x?xf32>)
{
    linalg.matmul  %arg0, %arg1, %arg2 :
      (memref<?x?xf32>, memref<?x?xf32>, memref<?x?xf32>)
    return
}
```

Figure 2: Two possible representations of a matrix multiplication in MLIR (left: using operations from the scf and std dialect; right: using the matmul operation from the linalg dialect)

Rebasing the **Deliverable 2.3** compiler to MLIR has two key advantages.

While, the schedule tree-based representation in the **Deliverable 2.3** compiler allowed for an accurate representation of matrix-vector products and matrix-matrix products, the representation is not guaranteed to be unique and reliable identification thus requires a non-trivial canonicalization pass prior to CIM-specific transformations. In an MLIR-based representation, these operations can be represented natively with operations from the linalg dialect. This significantly simplifies both the matching procedure, which can be reduced to the identification of sequences of dependent high-level linear algebra operations, and the transformation, which becomes a simple replacement of MLIR operations using the MLIR rewriting infrastructure. Also, transformations are no longer limited to the polyhedral fragment, since the rewriter allows for the replacement of any kind of operation.

Second, the use of MLIR leverages an extensive and extensible ecosystem, including a complete lowering paths from MLIR to machine code. This shifts the focus from middle-end and backend transformations to the development of an MLIR code generation scheme in the frontend and CIM-specific transformations. Finally, MLIR benefits from strong support by industry-leading companies and the LLVM community.

3 Overview of the Revised Compilation Flow

Figure 1 provides an overview of the updated compilation flow for **Deliverable 2.4**. As in the compilation flow from **Deliverable 2.3**, the source program is specified as a *Tensor Comprehension*, a high-level, abstract representation inspired from the Einstein notation of tensor algebra. The signature of a Tensor Comprehension specifies the element types, shapes and names of a set of input and output tensors. The body defines the operations to be performed on the input tensors and is composed of arithmetic expressions, tensor index expressions and assignments to output tensors. Upon entry in the compilation flow, the textual representation of a Tensor Comprehension is parsed by the *TC parser* extracted from the original *Tensor Comprehensions* project, which generates an abstract syntax tree (AST), labeled *TC lang*.

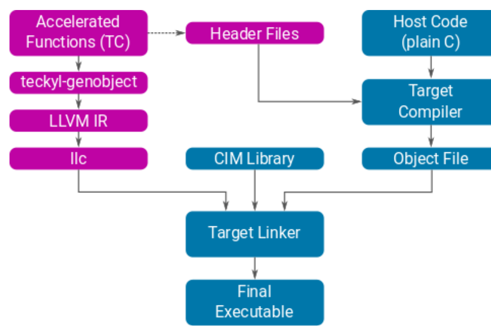

Figure 3: Compilation scheme combining accelerated functions, host code and the run-time library

The *Teckyl* frontend traverses this AST and generates an MLIR module with operations from different dialects suited to represent the tensor operations. High-level operations, such as matrix-vector products, matrix-matrix-products, tensor initializations with constant values, vector additions and row-wise matrix-vector additions are modeled as so-called *structured operations* representing specific linear algebra operations and generic tensor operations from the *linalg* dialect, respectively. The remaining tensor operations are modeled as loop nests of for loops from the *scf* (static control flow) dialect, with operations in the loop bodies from the *std* (standard) dialect, modeling element-wise arithmetic and memory accesses.

In a CIM-specific compilation pass, the Teckyl frontend examines the generated *linalg* operations and replaces sequences of dependent operations that can directly be mapped to the CIM accelerator with calls to functions from the CIM run-time. These are operations corresponding to biased matrix-vector products, modeled as a sequence of:

1. A *linalg.fill* operation initializing an output vector *o* with zero values
2. A *linalg.matvec* operation, multiplying a vector *a* with a matrix *M* and adding the result to the output vector *o*
3. A *linalg.generic* operation that adds the values a bias vector *b* to *o*

or biased matrix multiplications, modeled as a sequence of:

1. A *linalg.fill* operation initializing an output matrix *C* with zero values
2. A *linalg.matmul* operation, multiplying two matrices *A* and *B* and adding the result to the output matrix *C*
3. A *linalg.generic* operation that adds the columns of a bias vector *b* to the rows of *C*

The resulting MLIR module is then passed to the standard MLIR tools *mlir-opt*, lowering high-level operations from *linalg* and remaining *scf* loops first to *std* and then lowering *std* operations to the *llvm* dialect and *mlir-translate*, which converts the operations from the *llvm* dialect to LLVM IR. This leaves the execution of operations that do not fit the CIM-specific patterns to the general-purpose CPU of the host.

The result is compiled with the standard LLVM tool *llc* in order to generate an object file, which can then be linked with the CIM run-time library and host code into a final binary. For convenience, the *teckyl-genobject* compiler driver orchestrates the execution of of the above-mentioned tools with appropriate parameters with a single invocation.

Figure 3 shows the embedding of *teckyl-genobject* into the overall compilation scheme combining accelerated functions specified as Tensor Comprehensions, host code and the CIM run-time library. Accelerated functions are compiled using *teckyl-genobject* as outlined above and may include calls into the CIM run-time. Although teckyl-genobject can generate object code directly, in cross-compilation scenarios, where the machine compiling the code and the CIM host have a different architecture, it is often more convenient to generate LLVM IR that is then passed to *llc* with appropriate parameters.

For interoperability with the code executing on the general-purpose CPU of the host, Teckyl is also capable of generating C header files that allow for invocation of the accelerated functions from the host code. The host code referencing the generated header files is compiled with an appropriate general-purpose compiler for the target architecture and all object files are linked with the CIM run-time, resulting in the final binary.

4 Illustration of the Compilation flow on the HAR Kernel

We illustrate the compilation flow on the main computational kernel from the **Work Package 1** HAR application. The Tensor Comprehension at the core of this application models the three stages of the neural network in the file *har-static.tc* as follows:

```
def har_inference(int4(900) input,
                  int2(128, 900) W0,
                  int2(128, 128) W1,
                  int2(12, 128) W2,
                  int4(128) B0,
                  int4(128) B1,
                  int4(12) B2) ->
   (int4(128) hidden_zero,
    int4(128) hidden_one,
    int4 (12) output)
{
   hidden_zero(i) +=! W0(i, j) * input(j) where i in 0:128, j in 0:900
   hidden_zero(i) += B0(i) where i in 0:128

   hidden_one(i) +=! W1(i, j) * hidden_zero(j) where i in 0:128, j in 0:128
   hidden_one(i) += B1(i) where i in 0:128

   output(i) +=! W2(i, j) * hidden_one(j) where i in 0:12, j in 0:128
   output(i) += B2(i) where i in 0:12
}
```

The signature defines a kernel named *har_inference* with a vector for a single input of 900 values, three 2-bit weight matrices of size 128×900, 128×128, and 12×128, respectively, as well as three 4-bit bias vectors of size 128, 128 and 12 for the fully-connected neural network layers. Intermediate results for the two hidden layers are stored in the output parameters *hidden_zero* and *hidden_one* of size 128. The final result is provided in a 4-bit vector named *output* with 12 elements.

Each layer of the neural network is modeled as a matrix-vector product, followed by an addition of a bias vector. The product is implemented using the +=! operator, which

accumulates the scalar product on the right-hand side of the expression at the element specified of the left-hand side, starting with the neutral element zero.

When compiling the comprehension with the Teckyl frontend without the CIM-specific pass, e.g., using the following command:

```
$ teckyl -emit=mlir --specialize-linalg-ops --body-op=linalg.generic \
    har-static.tc
```

this yields the following representation in MLIR (only the code generated for the first two statements of the comprehension's body is shown, as the remaining statements yield similar code):

```
#map3 = affine_map<(d0) -> (d0)>

module {
  func @har_inference(
    %arg0: memref<?xi4>,
    %arg1: memref<?x?xi2>,
    %arg2: memref<?x?xi2>,
    %arg3: memref<?x?xi2>,
    %arg4: memref<?xi4>,
    %arg5: memref<?xi4>,
    %arg6: memref<?xi4>,
    %arg7: memref<?xi4>,
    %arg8: memref<?xi4>,
    %arg9: memref<?xi4>)
{
    %c0_i4 = constant 0 : i4
    linalg.fill(%arg7, %c0_i4) : memref<?xi4>, i4
    linalg.matvec  %arg1, %arg0, %arg7 :
    (memref<?x?xi2>, memref<?xi4>, memref<?xi4>)
    linalg.generic {
    args_in = 1 : i64, args_out = 1 : i64, indexing_maps = [#map3, #map3],
    iterator_types = ["parallel"]
    } %arg4, %arg7 {
          ^bb0(%arg10: i4, %arg11: i4):
          %0 = addi %arg10, %arg11 : i4
          linalg.yield %0 : i4
    }: memref<?xi4>, memref<?xi4>
    ...
    return
  }
}
```

The representation defines a single module with a single function with the same name as the source comprehension. The parameters *%arg0* through *%arg7* correspond to the input and output tensors of the comprehension. The type of each parameter is a shaped memory reference with either one or two dimensions of 4-bit and 2-bit integers.

For the first statement, Teckyl has generated a *linalg.fill* operation, initializing the output vector for the first hidden layer with the zero constant *c0_i4*, and a *linalg.matvec* operation, performing a matrix-vector product of the first two arguments and accumulating the result in the hidden layer. For the second statement, which adds the bias vector to the result of the first hidden layer, Teckyl has generated a *linalg.generic* operation, iterating over the output elements of the hidden layer and the elements of the argument *%arg4* (corresponding to the bias vector *B0*). The body of this operation is composed of a single basic block *^bb0*, adding two elements with an integer addition operation *addi* and returning the result with a *linalg.yield* operation.

The CIM-specific pass for pattern recognition is invoked by adding the command line parameter *–detect-cim-patterns.* This causes the biased matrix-vector products to be replaced with calls to a run-time function *cim_mv_2xi2_1xi4_1xi4_1xi4* (the suffix indicates the element types of the tensor operands):

```
module {
  func @har_inference(...) {
    call @cim_mv_2xi2_1xi4_1xi4_1xi4(%arg1, %arg0, %arg4, %arg7) :
          (memref<?x?xi2>, memref<?xi4>, memref<?xi4>, memref<?xi4>) -> ()
    call @cim_mv_2xi2_1xi4_1xi4_1xi4(%arg2, %arg7, %arg5, %arg8) :
          (memref<?x?xi2>, memref<?xi4>, memref<?xi4>, memref<?xi4>) -> ()
    call @cim_mv_2xi2_1xi4_1xi4_1xi4(%arg3, %arg8, %arg6, %arg9) :
          (memref<?x?xi2>, memref<?xi4>, memref<?xi4>, memref<?xi4>) -> ()
    return
  }

  // Declaration of the run-time function
  func @cim_mv_2xi2_1xi4_1xi4_1xi4(
    memref<?x?xi2>, memref<?xi4>, memref<?xi4>, memref<?xi4>)
}
```

This function must be implemented by the CIM run-time and perform the actual offloading to the device. The separation of the offloading function from code generation by the compiler allows for dynamic dispatching of operations at execution time if multiple CIM tiles are available. For example, in a configuration with a tile group of three tiles with data bypassing, the function could dispatch the work in a round-robin fashion to the different tiles:

```
#include <stdint.h>
#include <memref.h>
#include "cim.h"

void cim_mv_2xi2_1xi4_1xi4_1xi4(...)
{
  static int cim_group = 0;

  /* Transfer biases and weights */
  store_biases(cim_group, b_allocatedPtr, b_size0);
  store_gemm(cim_group, A_allocatedPtr, A_size1, A_size0, A_size0, 0, 0);
```

```
  /* Ensure that weights and biases have been transferred entirely */
  cim_spinlock(cim_group);

  /* Transfer inputs and indicate that the operation is a matrix-vector
   * product */
  cim_blas_batch(cim_group, x_allocatedPtr, o_allocatedPtr, 1, A_size0,
                 A_size1);

  /* Round-robin distribution to the tiles */
  if(++cim_group == 3) {
    /* Assignment to the last tile of the group triggers execution.
     * Synchronize only with the last tile; synchronization for
     * intermediate results is handled by the hardware. */
    cim_spinlock(2);
    cim_group = 0;
  }
}
```

Note that it is the responsibility of the run-time function to ensure the correct order of low-level operations, including synchronization with the CIM accelerator. Also, mixing code executing on the host CPU and the CIM accelerator requires careful verification by the programmer, since the host CPU uses wrapping arithmetic common for general-purpose cores, while the CIM accelerator uses saturating arithmetic.

The signature of the accelerated function required for invocation from host code can be generated with the option -emit=header:

```
$ teckyl -emit=header har-static.tc
```

This includes the signature of the MLIR function *har_inference* with parameters of the MLIR type memref, as well as a wrapper function named *har_inference_wrap* that can be called with ordinary pointers to continuous memory regions.

5 Building and using the Compiler

The **Deliverable 2.4** compiler comes as a modified version of the *Teckyl* project and is available from the TU/e gitlab server. This version of *Teckyl* depends itself on a modified version of the *llvm-project* repository, also hosted on the TU/e gitlab server. When cloning the git repository, please make sure to perform a recursive clone, e.g.:

```
$ git clone --recursive ssh://git@git.ics.ele.tue.nl/mnemosene/cim-compiler
```

To build the compiler and associated tools, the sources first need to be configured using the *cmake* build system:

```
$ cd cim-compiler
$ mkdir build
$ cd build
$ cmake ..
$ make -j teckyl mlir-opt mlir-translate llc
```

When the build process has finished, the build directory contains the *teckyl* binary in *bin* and the *mlir-opt*, *mlir-translate* and *llc* binaries in *llvm-project/llvm/bin*. In order to use the teckyl-genobject script, the paths to the binaries above must be made available through the *PATH* environment variable, e.g., with:

```
$ export PATH="$PWD/bin:$PWD/llvm-project/llvm/bin:$PATH"
```

executed from the build directory. Generating a file with LLVM IR named *example.ll* from a source file *example.tc* with a Tensor Comprehension is straightforward:

```
$ teckyl-genobject --mode=llvmir -body-op=linalg.generic \
    --detect-cim-patterns -o example.ll example.tc
```

The output can then be processed by *llc* and an appropriate cross-compiler driver invoking a cross-assembler with appropriate flags for the target architecture. For example, for the simulator, the following commands are used:

```
$ llc -float-abi=soft -mtriple arm-none-eabi -mcpu=cortex-a9 \
    -o example.S example.ll
$ arm-none-eabi-gcc -mcpu=cortex-a9 -mfloat-abi=soft -marm \
    -c example.S -o example.o
```

For a complete, end-to-end example from the source to simulation, a repository referencing the **Deliverable 2.4** compiler, the **Work Package 5** simulator and the HAR application from **Work Package 1** has been created. This can be cloned as follows:

```
$ git clone --recursive \
    ssh://git@git.ics.ele.tue.nl/mnemosene/cim-compiler-environment
```

Invoking the *make* utility with the default target at the root directory of the repository builds the compiler and the simulator:

```
$ cd cim-compiler-environment
$ make -j
```

Four versions of the HAR application are included in the repository:

- The default version, simply referred to as *har*, with 4 bit inputs, 2 bit weights and 4 bit biases, mapping each of the three layers of the neural network a different tile of a tile group of three tiles with data bypassing.
- A batched version of the default above, grouping inputs into a matrix and processing all inputs at once.
- A version executing on a single tile with a precision of 8 bits for inputs, weights and biases, referred to as *har-8bit-single-tile*
- A batched version of the single tile application

The applications can be found in the subdirectories *applications/har* and *applications/har-8bit-single-tile*, respectively. Each application comes with a *Makefile* that allows the user to

build the application with the **Deliverable 2.4** compiler and simulate it with the **Work Package 5** simulator, e.g., with the following commands:

```
$ cd applications/har
$ make
$ ./run.sh              # simulate the non-batched version
$ ./run.sh --batched    # simulate the batched version
```

6 Conclusions

The final version of the parallelizing compiler for the CIM architecture allows for the compilation of high-level representations of critical operations of applications identified in **Work Package 1** for the CIM architecture with a simplified and extensible compilation flow.

The end-to-end compilation flow and orchestration of accelerated functions with host-code and the CIM run-time has been verified with multiple implementations of the HAR application from **Work Package 1** (non-batched / batched version with mixed 4-bit / 2-bit precision for a tile group of size three, non-batched / batched version for an 8-bit single tile configuration) executed on the **Work Package 5** simulator.

The use of the MLIR multi-level intermediate representation allows for a more robust identification of performance-critical operations for offloading, simplified transformations and a simplified code generation scheme based on common MLIR and LLVM infrastructure.

References

[1] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. ACM Trans. Archit. Code Optim. 16, 4, Article 38 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355606

[2] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13). ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[3] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. In International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria.

[4] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. ArXiv preprint. https://arxiv.org/abs/2002.11054.