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

D2.1 – First version programming interface at the micro- and macro- levels

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:	TUE
Report Issue Date:	14/01/2019

Document History
(Revisions – Amendments)

Version and date	Changes
1.0 5/12/2018	First version
1.1 21/12/2018	Draft prepared
1.2 28/12/2018	Formatting improvements
1.3 31/12/2018	Formatting improvements

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

Table of Contents

1 Introduction 4
2 Micro-instruction set support 5
  2.1 CGRA architecture 5
    2.1.1 Architecture extensions 5
    2.1.2 Assembler support 6
    2.1.3 Compiler support 6
  2.2 PULP architecture 6
3 Macro-instruction set support 8
  3.1 Macro-instruction implementations on PULP and CGRA 8
    3.1.1 PULP 8
    3.1.2 CGRA 8
  3.2 Tensor comprehensions and polyhedral compilation 9
  3.3 Declarative polyhedral pattern recognition and transformation 10
4 Conclusions 12

1 Introduction

The need for energy efficient yet high performance compute capability has kept growing since the invention of digital signal processing. Each year, new inventions in computer architecture keep stretching the limits of those generations that came before. However, the complexity of these new systems makes programming them a challenge requiring expert knowledge of the internal workings of these systems. This makes their adoption often painful and many new architectures have failed in the past because they were simply too difficult to use in a production environment.

This is a risk we must also face when introducing the compute-in-memory (CIM) paradigm with the memristor processing elements[1]. For a part, the CIM aspects can be hidden from the programmer by the introduction of abstract instructions at the micro-level[2]. However, the usage of these instructions still requires that programmer to think about the smaller steps taken to form an application kernel and about the data movement required to get all operands into the CIM tile.

Within this project we distinguish three operation levels for the CIM hardware. At the nano-level we recognize individual memristor activities, mostly performed in the analog domain and abstracted away by the micro-level. The micro-level is the lowest level represented in the digital hardware surrounding the CIM tile. At this level basic operations on the array are possible but still considering detailed knowledge of the CIM hardware. For the programmer however, it may be more useful to think about operations at the macro-level. Where it is possible to reason about vector and matrix operations and entire application kernels can be represented. These macro-level operations can then be translated into the micro-level either through the usage of a support library, created by an expert programmer, or by direct translation through a compiler.

On top of the abstractions provided by the macro- and micro-level operations a compiler should be developed that can support the programmer by recognizing program sections which can be moved to the CIM hardware. Automatically managing the required data movement through the system and selecting the right instructions for the CIM hardware at either the macro- or micro-level. The introduction of such a tool would significantly improve the usability of the proposed CIM paradigm and should aid in its adoption. Furthermore, the ability to quickly introduce new applications running on the platform will also impact the quality of the design investigation as more, and more complex, benchmark applications can be mapped.

In this document we describe the current state of the programming tools for both the PULP and CGRA platforms as used in the MNEMOSENE project as basis for the CIM platform architecture. Overall, this document is organized as follows. Chapter 2 introduces the current compilation tools available for the CGRA and PULP platforms and how the introduction of micro-level instruction set impacts these tools. Chapter 3 then introduces the possibilities for recognizing macro-level kernel operations in high level code and how these can be manipulated to impact the required data transfers to and from the CIM tile. Chapter 4 then concludes this document.

[1] As described in Deliverable D4.1 and D4.2
[2] Introduced in Deliverable D3.1

2 Micro-instruction set support

The introduction of the CIM tile into the CGRA and PULP platforms presents new challenges and opportunities to the currently existing programming tools for both platforms. The following sections will describe for each platform the usage and required extensions to the tools.

2.1 CGRA architecture

The CGRA platform consists of a flexible hardware platform, illustrated in Figure 2-1. This template allows the user to connect different function units (FU) together using a reconfigurable interconnect. As such, it allows for the instantiation of custom processor architectures on the CGRA fabric, which can be specialized for individual applications.

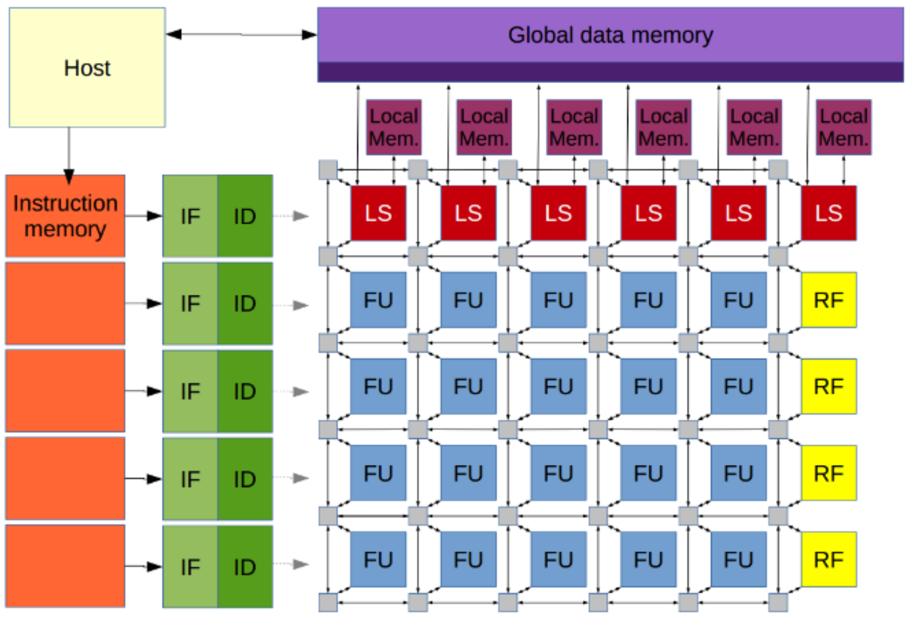

Figure 2-1: The Coarse Grain Reconfigurable Architecture (CGRA) platform template

In the MNEMOSENE context the CIM tile for the CGRA will be embedded into this platform template as an additional FU type. This allows for a tight integration between the data processing and parallel memory interfaces of the CGRA platform and the CIM tile.

2.1.1 Architecture extensions

As described above, the CIM tile will be added to the CGRA platform as a new FU type. To integrate this into the current hardware synthesis flow some extensions are required to the existing tools.

First of all the current architecture template is described to the tools as an XML description of the available platform parameters. Hardware instances can be generated directly from this description and will need to include the CIM tile in their considerations. Each individual FU in the CGRA instantiation has the option to be connected to its own instruction memory (IF/ID). Alternatively, multiple FUs of the same type may share a single IF/ID, effectively causing them to operate in a SIMD fashion.

For the CIM tile this means that it will get its own instruction stream containing the micro-level instructions for the CIM operations. The order of execution for these operations is under control of the CGRA data-path. Through this, it is possible to schedule macro-level kernels on the CGRA and have it manage the data flow into, and out of, the CIM tile.

2.1.2 Assembler support

The basic, low-level, programming interface to the CGRA is provided by a parallel assembler language (PASM). This language essentially describes the operations scheduled for each of the IF/ID paths connected to the FUs. At this level, the individual contents of the instruction memories can be clearly distinguished. Here, data is explicitly moved between load-store units (LS), function units, and register files (RF).

Adding the CIM tile to the architecture template will require the addition of the CIM micro-level instructions into the PASM language description, and to teach the assembler about the required instruction encoding.

2.1.3 Compiler support

The CGRA platform also provides an initial compiler framework for improved usability. This compiler framework is currently still in development but already provides the possibility to map high level code, currently using C as input language, onto a user specified processor instantiation on the CGRA fabric. To achieve this, the LLVM based compiler loads a description of the CGRA processor instantiation during the compilation process and uses this to perform the operation scheduling and mapping. As a result, high-level C code can be mapped onto the CGRA without requiring much in-depth knowledge of the platform.

Currently this compiler supports mapping of applications onto basic scalar architectures, optionally having multiple units of the same type available in a VLIW fashion. Extensions are in process to also support SIMD operations and to use the LLVM auto-vectorization features to automatically generate parallel program code on the CGRA. In parallel, work is in progress to extend the CGRA compiler to also include newer programming languages such as Halide.

For the CIM tile support the current compiler needs to be extended with knowledge on the CIM micro-level operations. Initially these will be mapped to compiler intrinsics as they do not have a direct representation in the abstract intermediate representation (IR) used within the compiler. Considering the data management requirements, the current CGRA compiler for scalar code is already sufficiently capable for managing the data transfers into, and out of, the CIM tile.

2.2 PULP architecture

The PULP platform integrates the CIM tile by adding it as a more traditional hardware accelerator. Figure 2-2 illustrates the connectivity of the PULP hardware together with the CIM tile.

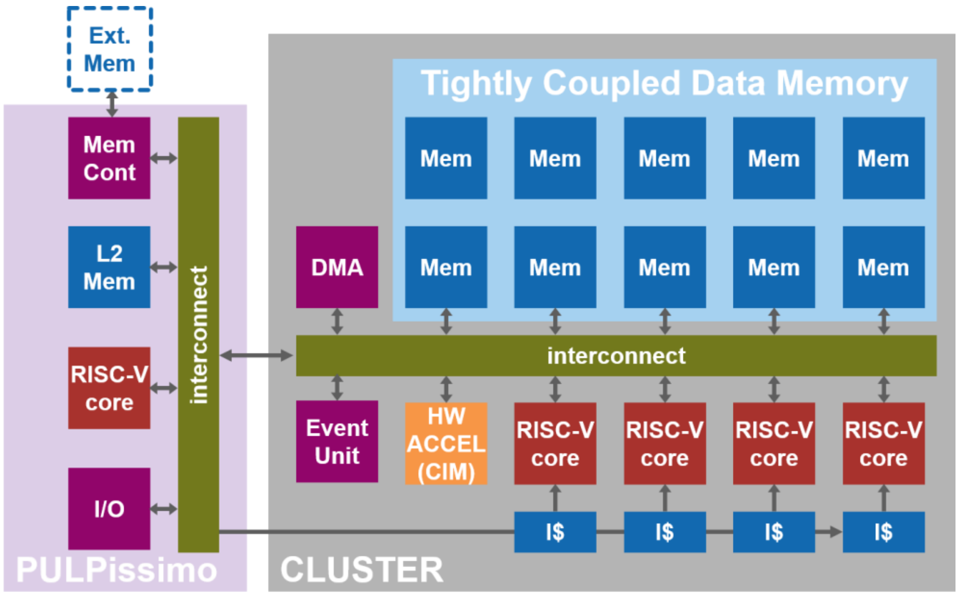

Figure 2-2: The parallel ultra-low power (PULP) platform

For configuration and synchronization purposes all the PEs are also connected to the independent so-called peripheral interconnect. This secondary interface makes it possible to prepare the CIM accelerator for autonomous operation by writing to memory-mapped configuration registers that store the desired memory access pattern and CIM instructions to be executed. Upon start or termination of the offloaded task the CIM accelerator may issue events using the event unit to synchronize with the general-purpose PEs for post processing or preparation of the next task.

In this case, the main task of the programmer will be to manage the high-level data transfers and configure the CIM tile for its autonomous operation. As such, this programming interface can be mostly abstracted from the user by providing a library interface to control the processing functions of the CIM tile. The use of such a library interface greatly simplifies the integration of the CIM tile into the platform but also limits the possibilities for optimization between kernel invocations. As data transfers are already pre-determined by the library it is no longer possible to keep data local to the CIM tile when it is used by subsequent kernel invocations without adding a specific implementation for such sharing into the library.

3 Macro-instruction set support

To the end user, the processing of high-level application constructs by the CIM tile is easier to relate to the overall application behavior. For this purpose, we present the macro-level operation interface to the CIM tile. At this level, the user can start reasoning about generalized data-processing algorithms and mainly needs to relate those to the application to be implemented.

There are two ways of integrating the CIM tile into a processing platform as illustrated by the PULP and CGRA platforms, each having its own approach of providing the user with a macro-level programming interface. This chapter will introduce both options and then presents a high-level programming approach that may be used to automatically identify kernel regions in a user-provided program and is able to optimize the data movement between subsequent kernel executions on a single CIM tile.

3.1 Macro-instruction implementations on PULP and CGRA

In general, there are two approaches possible when implementing macro-level operations onto a CIM platform. Either the platform vendor provides a support library, which implements the macro-level kernels using micro-level operations and provides this for use in higher level applications. Or, the platform compiler needs to learn how to do the translation of macro-level operations into the micro-level code generation. In the MNEMOSENE project the PULP platform is used to demonstrate the former (library-based), while the CGRA platform is used to demonstrate the latter (compiler based).

3.1.1 PULP

The library-based approach only requires the production of the actual library program code. In principle this requires no changes to the pre-existing compilation process. For the PULP platform this is further emphasized by the possibility of a stand-alone CIM tile which will run through its micro-level operations in an autonomous fashion. As a result, only code related to managing kernel preparation and data transfer setup towards the CIM is required to run on the RISC-V cores of the platform. These management procedures are easily implemented in a high-level library and will not impact the compilation process on the PULP platform.

3.1.2 CGRA

For the CGRA this mapping process is a bit more involved. Here the design is to make a more tightly coupled integration between the CIM tile and the CGRA fabric. A more tightly collaboration between the programming of both is thus also expected. The advantage of this is that more optimization possibilities are enabled when it comes to kernel fusion and data locality. However, this comes at the cost of added complexity in the programming environment.

For this purpose, the CGRA compiler will need to be extended with knowledge about the micro-level instructions, their behaviour and timing characteristics, and the ability to handle more complex operations needs to be introduced into the scheduling framework for the current compiler. Initial support for such more complex operations is in progress as it is also required for some of the SIMD extensions that are currently investigated for the CGRA platform.

With these extensions, it is then possible to provide the compiler with support for the CIM micro-level operation programming interface. Initially this will be through providing the micro-level instructions as intrinsics within the compiler. Later, this will be extended to also generate sequences of such micro-level operations from macro-level operations as recognized by the automatic framework described below.

3.2 Tensor comprehensions and polyhedral compilation

The Tensor Comprehensions project started as a collaboration between Inria and Facebook Artificial Intelligence Research (FAIR) in 2017. It is a Domain-Specific Language (DSL) focusing on deep learning, but it also provides a solid basis for the design of DSLs suitable for scientific computing, general tensor algebra, and signal and media processing. This section of the deliverable highlights the application of Tensor Comprehensions (TC) as a front-end DSL instantiating the macro-instruction programming interface in the deep learning domain. The next section will describe how TC, and in general, any polyhedral compiler, can be retargeted to the CIM tile micro-instructions.

Deep learning models with convolutional and recurrent networks are now ubiquitous and analyze massive amounts of audio, image, video, text and graph data, with applications in automatic translation, speech-to-text, scene understanding, ranking user preferences, ad placement, etc. Competing frameworks for building these networks such as TensorFlow, Chainer, CNTK, Torch/PyTorch, Caffe1/2, MXNet and Theano, explore different tradeoffs between usability and expressiveness, research or production orientation and supported hardware. They operate on a DAG of computational operators, wrapping high-performance libraries such as CUDNN (for NVIDIA GPUs) or NNPACK (for various CPUs), and automate memory allocation, synchronization, distribution. Custom operators are needed where the computation does not fit existing high-performance library calls, usually at a high engineering cost. This is frequently required when new operators are invented by researchers: such operators suffer a severe performance penalty, which limits the pace of innovation. Furthermore, even if there is an existing runtime call these frameworks can use, it often does not offer optimal performance for a user's particular network architecture and dataset, missing optimizations between operators as well as optimizations that can be done knowing the size and shape of data. Tensor Comprehensions include (1) a language close to the mathematics of deep learning, (2) a polyhedral Just-In-Time compiler to convert a mathematical description of a deep learning DAG into a CUDA kernel with delegated memory management and synchronization, also providing optimizations such as operator fusion and specialization for specific sizes, (3) a compilation cache populated by an autotuner. In particular, we demonstrate the suitability of the polyhedral framework to construct a domain-specific optimizer effective on state-of-the-art deep learning models on GPUs. Our flow reaches up to 4× speedup over NVIDIA libraries on kernels relevant to the Machine Learning Community, and on an actual model used in production at Facebook. It is integrated with mainstream frameworks Caffe2 (production-oriented), PyTorch (research-oriented), through the ATen asynchronous tensor library.

The full description and evaluation of TC are available in a research report and online blog, with links to the code and documentation:

https://research.fb.com/announcing-tensor-comprehensions.

The instantiation of the TC framework on MNEMOSENE CIM tiles quickly emerged as the most effective path to an end-to-end compilation flow, allowing domain experts to map machine learning models in their classical environment (PyTorch) to be accelerated through the MNEMOSENE tool flow and non-volatile memory architecture. The MNEMOSENE project supports the adaptation of TC to the CIM tile, but not the main infrastructure that was designed earlier and in parallel with the first six months of the project with industry support and funding. The port to CIM tiles is dependent on the macro-to-micro instruction compiler being completed and specialized for CIM micro-instructions, which is the purpose of the next section.

3.3 Declarative polyhedral pattern recognition and transformation

As described at the beginning of this document, the macro-instruction programming interface relies on a macro-to-micro instruction compiler, designed and implemented in WP2. This compiler is a first of a kind, bringing together:

1. a polyhedral compilation framework to automate the lowering and mapping of high level, domain-specific constructs, to a compute-in-memory architecture;
2. declarative, pattern-matching and substitution on a polyhedral representation suitable for the acceleration of a wide range of numerical computations.

This section summarizes the approach and highlight its application to the design of a compilation flow targeting CIM tiles. The declarative transformation framework is described in detail in the Inria research report RR-9243:

https://hal.inria.fr/hal-01965599v1.

The source code is available and will be updated continuously through the second year of the project:

https://github.com/ftynse/islutils.

The pattern matching and transformation framework was designed as a generic approach to design domain- and/or target-specific compilers based on a polyhedral representation. The Tensor Comprehensions compiler and the PPCG source-to-source compiler are a natural candidate for integrating such a declarative approach, to implement new optimizations and to support target-specific idioms. The MNEMOSENE project requires both: optimizations to manage non-volatile memory, loading data into CIM tiles and transforming the data layout for faster CIM operation, as well as direct support for CIM tile micro-instructions, supported as a target-specific set of builtin instructions.

The proposed framework enables pattern matching and declaratively express transformations on schedule trees, an internal representation of the polyhedral compiler of Tensor Comprehensions and PPCG. Unlike the majority of polyhedral approaches, pattern matching tree-rewriting transformations in our framework can be implemented and augmented using classical compiler construction technology applicable to other tree-like structures in the internal representation of a compiler.

Our approach allows us to easily express and compose a set of program transformations that use the polyhedral analyses but do not require the conventional combination of affine loop transformations, usually obtained by solving linear optimization problems. Complementary to existing loop transformation, it now becomes possible to declare, refine and reuse transformations for, e.g., data layout or device mapping. The specialization of the framework to CIM tiles non-volatile memory and micro-instructions is the purpose of the second year of the project.

Here is an illustrative example from the associated research report. The candidate loops for a tensor contraction corresponding to a generalized matrix product can be found implementing simple conditions on the schedule tree and access relations. We are looking for a three-dimensional permutable band with a single statement (leaf) featuring specific access patterns: at least three two-dimensional read accesses to different arrays, and a permutation of indices that satisfies the placeholder pattern [i, j]→[i, k][k, j]. This can be expressed by using a relation matcher and a set of callback functions as shown in the code sample below:

```
auto isGemmLike = [&](isl::schedule_node) {
    auto _i = placeholder(ctx);
    auto _j = placeholder(ctx);
    auto _k = placeholder(ctx);
    auto _A = arrayPlaceholder();
    auto _B = arrayPlaceholder();
    auto _C = arrayPlaceholder();

    /* Restrict the access relations to this subtree */
    reads = reads.intersect_domain(node.domain());
    writes = writes.intersect_domain(node.domain());
    auto mRead = allOf(access(_A, _i, _j),
                       access(_B, _i, _k),
                       access(_C, _k, _j));
    auto mWrite = allOf(access(_A, _i, _j));
    return match(reads, mRead).size() == 1 &&
           match(writes, mWrite).size() == 1;
};

auto matcher = band(_and(is3D, isPermutable, isGemmLike)), leaf());
```

For each match, one may then apply tiling for cache locality, or specifically for CIM tiles, to align the computation with the size of the frame of non-volatile RAM available to the coefficients of a constant input tensor, to be stored into a given CIM tile. Then, after applying the tiling builder, one may define and apply the second builder to create the micro-kernel, further tiling the points loops so that they fit into CIM micro-instruction (SIMD) registers; one may also optionally fully unroll the new innermost loops to enable instruction-level parallelism if the target implementation is a CGRA (see the micro-level interface introduction above; the PULP interface features asynchronous library calls instead).

```
isl::schedule_node node = /* obtain node, e.g., from a matcher */ ;
auto macroKernel =
    band([&]() { return tileSchedule(node, /*local memory size*/ ); },
        band([&]() { return swapDims(pointSchedule(node), -2, -1); }));

/*Apply the previous builder and get the child band.*/
node = macroKernel.insertAt(node.cut()).child(0);
auto microKernel =
    band([&]() { return tileSchedule(node, /*vectorization*/ ); },
        band([&]() { return unrollAll(pointSchedule(node)); }));
```

Based on this general framework, we will be able to implement a CIM-tile specific backend targeting the micro-level programming interface. We will focus on representative pointwise operations, tensor contractions and convolutions occuring in deep learning models and scientific computing, to demonstrate the computational and energy efficiency of the proposed architecture and technology.

4 Conclusions

In this document we have described the initial framework for mapping application code onto the CIM tile. The goal of this framework is to provide a (semi-) automatic mapping of larger application parts onto the macro- and micro-level instructions of the CIM architecture as provided through the programming interfaces of both the PULP and CGRA platforms. The aim of this is to provide a demonstration of how a CIM tile can be used within the context of a larger system, as well as, to show the performance and energy impact of using the CIM paradigm at the system level view. This demonstration will take the form of the end-to-end acceleration of numerical computations expressed in the Tensor Comprehensions language. This end-to-end flow is made possible thanks to a existing polyhedral compilation flow, embedded into the Tensor Comprehensions compilation flow, and to a high-level framework for detecting the presence of kernels within a larger application based on a structural analysis of the application code. This design will enable us to validate the effectiveness of the proposed CIM technology and architecture in a realistic software environment, by automatically identifying program parts that may be suitable for execution on the CIM tile, as well as, generating the required data movement operations required to prepare the data for processing using the CIM operations.