

DATAFLOW ARCHITECTURES TO ACCELERATE EXECUTIONS OF INTERDEPENDENT OPERATIONS SUCH AS INVOLVED IN AI INFERENCES

TECHNICAL FIELD

The invention relates in general to the field of data processing devices for executing interdependent operations, in particular operations involved in neural network-based inferences, and methods for executing such operations. In particular, it is directed to a data processing device, which includes a control unit (e.g., a central processing unit) and several processing elements, as well as a data interconnect, a control interconnect, and a token interconnect. The token interconnect interfaces several processing elements, which can transmit tokens in accordance with conditions defined by commands dispatched by the control unit through the control interconnect. The commands specify conditions for executing operations in accordance with a predetermined interdependence of said operations. The commands can be accumulated at the processing elements. The correct orchestration of the executions is achieved thanks to the tokens sent and/or received via the token interconnect.

BACKGROUND

Artificial neural networks (ANNs) such as deep neural networks (DNNs) have revolutionized the field of machine learning (ML) by providing unprecedented performance in solving cognitive tasks. ANN operations involve matrix-vector multiplications (MVMs), which account for 70 to 90% of the total neural network operations, irrespective of the ANN architecture. MVM operations pose multiple challenges, because of their recurrence, universality, compute, and memory requirements.

Traditional computer architectures are based on the von Neumann computing concept, according to which processing capability and data storage are split into separate physical units. Such architectures suffer from congestion (the “von Neumann bottleneck”) and high-power consumption, as data must be continuously transferred from the memory units to the control and arithmetic units through interfaces that are physically constrained and costly.

One possibility to accelerate MVMs is to use dedicated hardware acceleration devices, such as in-memory compute (IMC) circuits having a crossbar array structure. This type of circuit includes input lines and output lines, which are interconnected at cross-points defining cells.

The cells contain respective memory devices, which are designed to store respective matrix coefficients that represent the ANN weights. Vectors are encoded as signals applied to the input lines of the crossbar array to perform the MVMs by way of multiply-accumulate (MAC) operations. This architecture can map MVMs simply and efficiently. The weights are updated (i.e., replaced) by reprogramming the memory elements, which makes it possible to repeatedly perform MVMs based on different weights. Such an approach breaks the “memory wall” as it fuses the arithmetic- and memory unit into a single IMC unit, whereby processing is done much more efficiently in or near the memory (i.e., the crossbar array). Finally, an IMC unit addresses the Von-Neumann Bottleneck on the instruction interface, inasmuch as a single instruction may suffice to operate MVMs over multiple cycles.

Such devices are particularly suited for accelerating the execution of ANNs on inferencing. Now, in addition to improving hardware alone, various acceleration strategies can be contemplated on the data processing level. For example, one may envisage the use of specific dataflow architectures (https://en.wikipedia.org/wiki/Dataflow_architecture), which allow multiple processing elements to work in parallel; task executions are determined by the availability of input data. However, the resulting dataflows are often extremely hard to program and control in practice, if only because of issues in terms of compilation and online data-dependency resolution.

Therefore, hardware accelerator architectures are needed, which allow to simply map interdependent operations on the hardware, while allowing a simple programming and control.

SUMMARY

According to a first aspect, the invention is embodied as a data processing device for executing interdependent operations. The device comprises a control unit, a memory device, several processing elements (PEs), and distinct interconnects. The PEs are configured to execute operations, e.g., matrix-vector multiplication (MVM) operations as involved in artificial neural network (ANN) executions. The interconnects include: a data interconnect, which interfaces the memory device with at least some of the PEs; a control interconnect, which interfaces the control unit with each of the PEs; and a token interconnect, which interfaces each of the PEs with at least another one of these PEs. The control unit is configured to dispatch commands to the PEs through the control interconnect. The commands specify conditions for executing the operations in accordance with a predetermined interdependence of the operations. In addition,

the PEs are configured to send and/or to receive tokens via the token interconnect, in accordance with said conditions, to orchestrate the executions of the operations. In operation, executing these operations causes at least some of the PEs to read from and/or write to the memory device through the data interconnect.

The PEs and interconnects of the data processing device give rise to a dataflow processing architecture that allows interdependent operations to be easily mapped onto the PEs. Remarkably, the PEs can signal each other by sending tokens to one another over the token interconnect. Such an interconnect is easy to fabricate. The token interconnect serves as a dedicated interconnect: it is not used for transmitting data or commands and other control signals; this is the role of the control and data interconnects. As a result, the token interconnect is available for fast token transmissions and allows fast signalling. The tokens are small information chunks (e.g., single bit values) that are transmitted in accordance with conditions as specified by the dispatched commands, with a view to enforcing a correct order of execution of the operations, taking into account data dependencies in the execution of the operations. The dispatched commands can be queued at the PEs to minimize the downtime at the PEs; the correct synchronization is ensured by the tokens. Moreover, the proposed architecture is easily scalable; data processing devices as described above can easily be clustered in a data processing system, which concerns another aspect of the invention.

Note, the execution dependencies (i.e., the dependencies between the executions of the operations) mostly depend on data dependencies (i.e., dependencies between data produced by some PEs and the data consumed by other PEs). In the present context, such execution dependencies are resolved at runtime by the tokens sent and received via the token interconnect. A simple example of dependency resolution is the following. The PEs may notably include a first PE and a second PE, which may be configured to execute a first operation and a second operation, respectively, in accordance with a first command and a second command, respectively, where such commands are dispatched by the control unit, in operation. Now, in embodiments, the second command is designed so that an execution of the second operation by the second PE is conditional on receipt by the second PE of an execution token transmitted by the first PE via the token interconnect upon completing the first operation. I.e., the second operation starts executing upon the second PE receiving the execution token, which ensures a correct execution dependency.

The first operation may for instance be designed to cause the first PE to write first data to the memory device through the data interconnect. Conversely, the second operation may be a data

load operation, designed to cause the second PE to read at least part of the first data from the memory device through the data interconnect.

In embodiments, one or more of the PEs include, each, a load interface unit and an execution interface unit. Each of the load interface unit and the execution interface unit is configured as a dual interface system, i.e., an interface unit that is connected to each of the token interconnect and the control interconnect. This way, each PE may proceed according to interdependent load and execute operations, where the execute operation is conditional on the load operation, itself conditional on an execute operation previously performed by a distinct PE. For example, a given PE may be configured to perform MVMs. This PE may have a load interface unit and an execution interface unit, whereby this PE can easily be instructed and controlled to trigger the execution of an MVM operation, provided this PE has first loaded relevant ANN weights (MVM load operation) to perform this MVM operation. Moreover, this MVM operation may have to use input vectors as obtained from activations on output vectors obtained from a previous MVM operation, as performed by a distinct PE. Separate load and execution interface units allow tokens to be optimally channelled and directed to orchestrate interdependent loading and execution operations at the PEs.

The data processing device may advantageously comprise a software endpoint component, which is connected to the token interconnect so as to interface the control unit with each of the PEs through the token interconnect. The software endpoint component may for example be integrated in the control unit. This provides additional flexibility as it allows the control unit to produce and consume tokens, such that the control unit can further interact with the PEs over the token interconnect. In particular, this makes it possible to sync interactions between the control unit and the PEs over the token interconnect: any task (e.g., storing/reading data on/from the memory device, exerting top level control, performing diagnostics) executed on the control unit can easily be synced with the PEs.

In embodiments, at least some of the operations include MVMs. In this case, the conditions specified by the commands cause said executions to result in executing neural layers, or portions thereof, of an ANN. That is, the data processing device is configured to execute the ANN. Interestingly, the conditions can be specified in the commands such that they cause portions of the neural layers to be executed in a depth-first manner. Each of these portions corresponds to one or more of the operations evoked above, whereby portions of the neural layers are executed in an alternating fashion, in operation. This reduces the memory requirements for the intermediate tensors between the alternating executed layers.

As noted above, at least some of the operations may include MVMs. In this case, at least one of the PEs may advantageously include an in-memory compute (IMC) unit, i.e., a processing unit that has a crossbar array structure, which is particularly suited for performing such MVMs. The crossbar array structure preferably includes $N \times M$ cells comprising respective memory systems, each designed to store K weights, where $K \geq 1$. That is, the crossbar array structure includes $N \times M$ memory systems, which are adapted to store K sets of $N \times M$ weights. Preferably, $K \geq 2$ (e.g., $K = 4$). In this case, the IMC unit may advantageously include a selection circuit, which is configured to enable $N \times M$ weights as active weights by selecting, for each of the memory systems, a weight from its K weights and setting the selected weight as an active weight. Thus, weight values can be locally switched at the IMC unit, without having to first transmit the corresponding data, i.e., without idle times. Still, new weight values may possibly be proactively loaded to avoid or reduce stall times (they can be loaded while the IMC unit executes based on the currently active $N \times M$ weights).

In addition, or in variants to IMC units as described above (where $K \geq 2$), the PEs may include simpler IMC units, which are configured to store a single set of weights at a time (i.e., $K = 1$). And, as noted earlier, IMC units may advantageously involve load and execution interface units to process tokens and accordingly sync operations, to timely load the weights and execute MVMs.

In addition, or in variants, to IMC units, other types of PEs may be involved. For example, the PEs may include one or more systolic arrays and/or one or more vector processing units. More generally, the present data processing devices may accommodate various heterogeneous types of PEs.

The transfer of data, commands, and/or tokens, is advantageously handled in a first-in-first-out (FIFO) manner, where possible.

This notably applies to data streamed from one PE to the next, should such PEs be connected through a data streaming interface. The PEs can be regarded as forming ordered pairs, inasmuch as information flowing from one element of a pair to the other likely differs from the information coming from the other element of this pair. Now, one or more ordered pairs of the PEs may advantageously be connected through a data streaming interface, which is preferably implemented as a FIFO channel. As said, not all of the PEs need to be interfaced with the memory device through the data interconnect. For example, at least some of the ordered pairs of PEs that are connected through a data streaming interface may not be interfaced with the

memory device through the data interconnect. Data streaming interfaces make it possible to move data from one PE to another without having to write/load data to the local, shared memory.

In addition, the control interconnect may advantageously interface with each of the PEs through a buffer designed as a FIFO queue. I.e., each buffer is designed to queue commands sent by the control unit in such a way that the commands are processed in a FIFO manner by each PE, in operation. Thus, commands can be accumulated at each PE. Each corresponding operation is executed as soon as this PE becomes available for execution, the tokens permitting, to reduce idle times.

Moreover, the token interconnect interfaces pairs of PEs, which can again be regarded as forming ordered pairs over the token interconnect. So, the PEs and the token interconnect may be jointly configured to allow unidirectional communication channels between these ordered pairs, where each of the unidirectional communication channels behaves as a FIFO channel. Thus, tokens can be passed and processed in a FIFO manner, whereby tokens do not need to encode the addressee (i.e., the address of the recipient PE). As a result, tokens can be encoded as mere bit values, as discussed later in detail.

Preferably, the data processing device further includes hardware-implemented counters, which are configured to monitor respective channels of the unidirectional communication channels. The counters increment or decrement respective counter values depending on whether tokens are sent or consumed through the respective channels. In this way, appropriate action can be taken should the maximum capacity of a channel be exceeded, to avoid deadlocks. This, however, can easily be avoided by provisioning sufficient channel capacities.

In embodiments, the data processing device is configured to operate the token interconnect for the latter to enable a two-signal ready/valid handshake interface between pairs of PEs. Note, for PEs having distinct load and execution interface units, any of these interface units may be connected over the token interface.

In preferred embodiments, the data processing device further includes a data movement engine (DME), which is connected to each of the control interconnect, the token interconnect, and the data interconnect. The DME is preferably implemented as a direct-memory access (DMA) controller. The DME allows data to be moved from one memory location to another, independently of the control units. Connecting the DME to the token interconnect allows the

data movements performed by the DME to be coordinated with operations performed by the PEs.

The data processing device will preferably be designed as an integrated circuit. Preferably, the memory device, the control unit, and the PEs, are all co-integrated in this integrated circuit.

According to another aspect, the invention is embodied as a data processing system, which includes a set of data processing devices such as described above. In addition, the data processing system includes a system-level memory device, a system-level control unit, and a system-level data interconnect, where the latter interfaces the system-level memory device with each of the data processing devices and the system-level control unit. The data processing system optionally includes a system-level DME, which is connected to the system-level data interconnect. Moreover, the data processing system optionally includes a system-level token interconnect, which is connected to each of the system-level control unit, the system-level DME, if any, and each of the data processing devices.

According to another aspect, the invention is embodied as a method of executing interdependent operations. The method relies on a data processing device as described above. I.e., the data processing device includes a control unit, a memory device, PEs, and distinct interconnects, namely a data interconnect, a control interconnect, and a token interconnect as described earlier. As noted earlier, at least some of the operations may include MVMs, while at least one of the PEs may include an IMC unit, which has a crossbar array structure.

The method revolves around operating the data processing device by operating the control unit and the PEs in accordance with the principles discussed above. That is, the control unit is operated so as to dispatch commands to the PEs through the control interconnect. Again, the commands specify conditions for executing the operations in accordance with a predetermined interdependence of the operations. The PEs are operated to send and/or receive such tokens via the token interconnect, in accordance with said conditions, to orchestrate the executions and accordingly cause at least some of the PEs to read from and/or write to the memory device through the data interconnect.

In embodiments, the control unit is operated in such a manner that the commands are dispatched by repeatedly backfilling the commands over the control interconnect, so that commands can be queued at the respective PEs.

In embodiments, the token interconnect enables unidirectional communication channels (i.e., token channels) between ordered pairs of the processing elements. Such unidirectional

communication channels preferably work as FIFO channels. In this case, the method may further comprise, for each unidirectional communication channel, incrementing (respectively decrementing) a counter value each time a token is sent (respectively consumed) through the respective channel. In addition, the method monitors the counter value against a threshold value, which is related to a token capacity of the respective channel.

In embodiments, the method further comprises upstream steps, which are performed prior to operating the data processing device. Such steps revolve around compiling execution instructions to identify the interdependence of operations and determine corresponding token dependencies. Such execution instructions capture early versions of the commands, e.g., before the token information is included in the commands. Eventually, the compilation causes to generate commands in accordance with the token dependencies determined. Such commands specify conditions that reflect the token dependencies determined by the compilation. At runtime, the control unit may dispatch the commands as obtained through the compilation.

The execution instructions will preferably be compiled under the constraint that the control unit is to be operated according to a linear execution schedule. In this case, the control unit can subsequently be operated according to the linear execution schedule. A linear execution schedule results in that the control unit dispatches one command after the next, whereby the dispatched commands queue at the PEs. A linear schedule is possible because the interdependence of the operations referred to in the commands can be resolved through the token interconnect.

In embodiments, the data processing device is operated to execute an ANN; at least some of the operations are assumed to include MVMs in this case. The conditions specified by the command cause the executions to result in executing neural layers, or portions thereof, of the ANN. Preferably, the data processing device is operated to execute the neural layer parts in a depth-first manner, in accordance with the commands sent to respective PEs, where each of the neural layer parts corresponds to one or more of the operations. The neural layer parts are orchestrated thanks to the tokens sent and received by the PEs via the token interconnect, such that portions of the neural layers are executed in an alternating fashion.

In practice, the token interconnect may interface each PE with μ components of the data processing device, where $\mu \geq 1$. A minima, the μ components include at least another one of the PEs. In other words, each PE is connected to at least another PE over the token interconnect. The commands are preferably structured as follows. Each command includes: data determining

an operation to be performed by a given PE (i.e., the PE that receives this command); μ bit fields determining up to μ tokens to be respectively received by the given element from the μ components before executing the operation; and μ bit fields determining up to μ tokens to be respectively sent by the given element to the μ components upon completion of the operation. This data structure ensures a compact representation of the commands, something that is possible thanks to the proposed hardware architecture.

In embodiments, μ is larger than, or equal to, $\nu - 1$, where ν corresponds to the total number of PEs that are interfaced with one another through the token interconnect. This means that the number of bit fields (in transmission and reception) is at least as large as $\nu - 1$, which corresponds to the number of connected PEs. That is, the token interconnect interfaces each PE with each of the $\nu - 1$ remaining PEs.

In particular, μ may be equal to ν , should the data processing device further comprise a software endpoint component. As noted earlier, the latter is preferably integrated in the control unit. This software endpoint component is connected to the token interconnect so as to interface the control unit with each of the PEs through the token interconnect. In addition, the token interconnect interfaces each PE with the control unit. I.e., μ is equal to $\nu - 1 + 1$.

The above examples assume that each PE can receive tokens from one or more other PEs and transmit tokens to one or more other PEs. The command structure can be further adapted to cases where the PEs include separate load and execution interface units.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other objects, features and advantages of the present invention will become apparent from the following detailed description of illustrative embodiments thereof, which is to be read in connection with the accompanying drawings. The illustrations are for clarity in facilitating one skilled in the art in understanding the invention in conjunction with the detailed description. In the drawings:

FIG. 1 is a diagram that schematically illustrates selected components of a data processing device that includes several processing elements and three different types of interconnects, as in embodiments. This results in a scalable dataflow architecture, which is capable of efficiently orchestrating executions of interdependent operations;

FIG. 2 is a diagram illustrating how data processing devices such as shown in FIG. 1 can be clustered to form a data processing system, as in embodiments;

FIG. 3 is a diagram illustrating how a linear task schedule of interdependent operations can be reordered from a 'layer-by-layer' type of execution to a 'depth-first' execution, mapped to processing elements of a data processing device such as shown in FIG. 1, as in embodiments. Such an approach can notably be applied to portions of layers of artificial neural networks, whereby portions of the neural layers can be executed in an alternating fashion, as in embodiments;

FIGS. 4 and 5 are diagrams illustrating variants to the data processing device of FIG. 1, according to embodiments;

FIG. 6A is a diagram illustrating a purposely simpler variant to the data processing device of FIG. 1, still according to embodiments. FIG. 6B shows a simple dependency graph that can be mapped onto the three processing elements of the data processing device of FIG. 6A, as in embodiments. FIG. 6C illustrates how commands can be accumulated at the processing elements and how the processing elements can signal each other by sending/receiving tokens through a token interconnect, where the tokens are coded through respective bitfields in the commands, with a view to orchestrating executions of operations in accordance with data dependencies thereof, as in embodiments;

FIG. 7 is a high-level schematic of a processing element configured as a hardware accelerator device, which includes an in-memory compute (IMC) unit, as in embodiments. FIG. 8 illustrates the crossbar array structure of the IMC unit of FIG. 7, where the crossbar array structure contains an array of $N \times M$ cells designed to store four sets of $N \times M$ weight values,

as in embodiments. The four sets of $N \times M$ weight values map four 2D arrays of $N \times M$ weight values, as illustrated in FIG. 9; and

FIGS. 10 and 11 are flowcharts illustrating high-level steps of a method of executing interdependent operations, using a data processing device such as shown in FIG. 1, according to embodiments.

The accompanying drawings show simplified representations of devices or parts thereof, as involved in embodiments. Technical features depicted in the drawings are not necessarily to scale. Similar or functionally similar elements in the figures have been allocated the same numeral references, unless otherwise indicated.

Devices, systems, and methods, embodying the present invention will now be described, by way of non-limiting examples.

DETAILED DESCRIPTION OF EMBODIMENTS OF THE INVENTION

Inferences performed by artificial neural networks (ANNs) are a data-bound compute workload, which potentially requires bringing a huge amount of data (e.g., input vectors, activations, weights) from high-capacity memories to the compute units. As the present inventors observed, exploiting data locality to prevent or reduce data movements is key to improving throughput and efficiency. On inferencing, ANNs act as complex function approximators, taking in multidimensional input data (arrays) and producing outputs that can again be multidimensional arrays. This approximation can be described as a compute graph, where the edges represent multidimensional tensors of intermediate results and the nodes represent neural layers, which perform compute operations on multidimensional tensor data.

Traditionally, these compute graphs are executed layer-by-layer, starting from the input data, by executing one layer at a time until one or more resulting tensor(s) are obtained. This execution model requires massive data traffic in the processing system: The size of the intermediate tensors can typically exceed megabytes for modern workloads. And this usually exceeds the size of lower-level caches and buffers in the system. In practice, this typically means that this data must be loaded and stored from higher-level memories or even external memory, which affects both the throughput and efficiency. More generally, similar issues can

be noted for workloads involving a large number of interdependent operations. It is therefore desirable to devise solutions for improving throughput and efficiency.

With this in mind, the present inventors have developed solutions to accelerate the execution of interdependent operations such as required to perform ANN inferences, as described in detail in the following.

The detailed description is structured as follows. General embodiments and high-level variants are described in section 1. Section 2 addresses particularly preferred embodiments. Section 3 concerns technical implementation details. Note, the present method and its variants are sometimes collectively referred to as the “present methods”. All references *S_n* refer to methods steps of the flowcharts of FIGS. 10 and 11, while numeral references pertain to devices, components, and concepts involved in embodiments of the present invention.

1. General embodiments and high-level variants

1.1. Data processing device

A first aspect of the invention concerns a data processing device for executing interdependent operations. FIGS. 1, 4, 5, and 6A, show examples of such a data processing device 10 – 13.

1.1.1. Essential features of data processing devices

Basically, the data processing device 10 – 13 includes a control unit 150, a memory device 180, processing elements (PEs) 100 – 105 (meant to execute operations), and distinct interconnects 110, 120, 170. The interconnects notably include a data interconnect 170, which interfaces the memory device 180 with at least some of the PEs. For example, the data interconnect 170 interfaces the memory device 180 with only a subset of the PEs 100 – 103, 105 in the example of the device 10 shown of FIG. 1; the PE 104 is not interfaced with the memory device 180 in this case, for reasons that will become apparent later. Conversely, the data interconnect 170 interfaces the memory device 180 with all the PEs 100a, 101a, 102 in the example of the device 11 shown in FIG. 4.

In addition, a control interconnect 110 interfaces the control unit 150 with each of the PEs 100 – 105. Finally, a token interconnect 120 interfaces each PE with at least another PE, a subset of PEs, or possibly all of the other PEs. The token interconnect 120 preferably interfaces each PE with all of the other PEs, as explicitly shown in FIG. 6C (corresponding to the data processing device 13 of FIG. 6A).

The PEs and interconnects of the data processing device give rise to an efficient dataflow processing architecture, which allows interdependent operations to be easily mapped onto the PEs. In detail, the control unit 150 is configured to dispatch commands to the PEs 100 – 105 through the control interconnect 110. Such commands specify conditions for executing desired operations in accordance with a predetermined interdependence of these operations. This interdependence is typically identified during a previous compilation of the operations. Besides, the PEs 100 – 105 are configured to send and/or to receive tokens via the token interconnect 120, in accordance with conditions as specified in the dispatched commands.

So, the control interconnect 110 enables a control path between the control unit 150 and the PEs, while the data interconnect 170 allows the PEs to access data from, or write data to, the memory device 180. I.e., executing the operations causes at least some of the PEs to read from and/or write to the memory device 180 through the data interconnect 170, in operation. Remarkably, the token interconnect 120 is used by the PEs to signal each other, which makes it possible to efficiently orchestrate the executions of the operations. Such an interconnect is easy to fabricate. It serves as a dedicated interconnect, which allows fast token transmissions and, therefore, fast signalling. I.e., as the token interconnect 120 is not used for transmitting commands and other control signals, it is available for rapid token transmissions. As one understands, this approach addresses issues of the von Neumann bottleneck.

The token interconnect effectively forms a network; it is sometimes referred to as a “token network” or “token network interconnect” in this document. In operation, the PEs signal each other by sending tokens to one another over the token interconnect 120. Tokens can be regarded as information chunks, which are transmitted in accordance with conditions specified by the dispatched commands, with a view to enforcing a correct order of execution of the operations, taking into account data dependencies in the execution of the operations.

Moreover, the proposed architecture is easily scalable, and data processing devices as described above can easily be clustered in a data processing system, as discussed later in reference to another aspect of the invention.

The main role of the control unit 150 is to dispatch commands. In principle, the control unit may be implemented as a processor or a set of processors. Preferably, the control unit 150 is implemented as a central processing unit (CPU) of the data processing device 10 – 13, as assumed in the accompanying drawings and in the following. In addition to sending the commands to the PEs through the control interconnect 110, the CPU 150 may for instance

communicate with the memory device 180, or any other memory device (e.g., a dedicated memory device, such as a cache memory, or an external memory device) to access the commands and then dispatch them. The control unit preferably integrates a cache memory caching data and instructions for better performance; the cache memory is assumed to form part of the CPU 150 in the accompanying drawings. The device 180 is preferably configured as a shared, multi-bank memory. The CPU may further communicate with the PEs through the token interconnect 120, as in embodiments described later.

Besides the individual processes (operations) to be performed by the PEs, the commands specify the conditions according to which such processes must be executed. To that aim, the commands can for instance be marked with tokens (i.e., they may include token identifiers, as illustrated in FIG. 6C), in such a manner that a transmitted token identifies the transmitting PE, as discussed later in detail. More generally, the commands may include instructions, attributes, and/or parameters, which define token dependencies that ensure a correct orchestration of the execution of the operations by the PEs. In other words, the conditions specified by the commands determine token dependencies that reflect the intended execution dependencies.

The tokens are preferably transmitted in a peer-to-peer manner, i.e., from one PE to the other, through the token interconnect 120. In this case, a connection between any two PEs through the token interconnect 120 is ensured by a dedicated channel, also referred to as a 'token channel' in this document. Each token channel is preferably implemented as a first-in-first-out (FIFO) channel. In variants, the token interconnect may include, or be connected to, a central component that receives all issued tokens and redispach them appropriately through the token interconnect. Preferred, however, is to rely on peer-to-peer connections, where each token channel connects a pair of PEs, for efficiency reasons.

The commands can be regarded as being sent asynchronously to the PEs, inasmuch as distinct PEs may receive respective commands in an order that does not necessarily reflect the relative order in which the underlying commands must be executed by the respective PEs. However, the PEs are preferably interfaced with the control interconnect through a buffer working as a FIFO queue, such that two commands successively sent to a same PE will be executed in the order the commands were received by this PE. In other words, the commands sent by the CPU can be sent asynchronously for any two distinct PEs but are sent in a prescribed order for each individual PE. The commands can thus be accumulated at the PEs, to minimize the downtimes at the PEs.

The data processing devices 10 – 13 are preferably manufactured as integrated circuits, where all components are preferably co-integrated in a same chip. In particular, the memory device 180, the CPU 150, and the PEs 100 – 105, may all be co-integrated in an integrated circuit. Interestingly, the present data processing devices can be designed as core computing devices (also referred to as ‘cores’ in this document), which can easily be clustered in a system 1 (see FIG. 2) to accelerate computations of complex interdependent operations, such as involved in modern ANN inferences (e.g., inferences made by transformer networks). In that case, the memory devices 180 can still be accessed locally by the PEs; each memory device 180 is typically configured as a local, shared multi-bank memory. Similarly, a cluster 1 of core computing devices 10 – 13 can possibly be manufactured as an integrated circuit, too, where all components of each of the core computing devices 10 – 13 can be co-integrated in a same chip. I.e., the proposed hardware architecture is easily scalable.

1.1.2. Examples of interdependent operations (load and execute operations)

All this is now described in reference to particular embodiments of the invention. Consider for example a task split into two operations (a first operation and a second operation) meant to be executed by two PEs (e.g., a first PE and a second PE), respectively. There, commands can be designed and sent by the CPU to these two PEs to require executions of the two operations, in a correct order. To that aim, the first PE may be required to send a token upon completing the first operation, while the second PE must receive this token prior to starting the execution of the second operation. The same principle can be applied step by step: each PE may be required to receive a token before starting to execute an operation and then transmit a token upon completing this operation. Thus, tokens sent and received by the PEs via the token interconnect make it possible to duly orchestrate the executions of the operations in accordance with a predetermined interdependence of the operations, e.g., as resulting from a previous compilation step. Examples of such operations are matrix-vector multiplications (MVMs), matrix-matrix multiplications, and dot products of vectors.

Let us illustrate this with a concrete example with reference to FIG. 1, where the PEs 100 – 105 include a first PE 100 and a second PE 101. The PEs 100, 101 are configured to execute a first operation and a second operation, as respectively specified in a first command and a second command dispatched by the CPU 150. Now, the second command can be designed so that the execution of the second operation by the second PE 101 is conditional on receipt by the second PE of an execution token transmitted by the first PE 100 via the token interconnect 120; this execution token is transmitted upon completing the first operation. This makes it possible to

ensure that the execution of the second operation will only be started upon completion of the execution of the first operation.

The above example illustrates how it is possible to ensure correct execution dependencies, where such execution dependencies reflect data dependencies. The first operation may for instance be designed to cause the first PE to write first data to the memory device 180 through the data interconnect 170. Conversely, the second operation may be a data load operation, designed to cause the second PE to read at least part of the first data from the memory device 180 through the data interconnect 170.

Each PE may actually proceed according to interdependent load and execute operations, where the execute operation is conditional on the load operation, itself conditional on an execute operation previously performed by a distinct PE. Other scenarios are described below, as well as in section 2. Note, the above examples assume that data is loaded from the memory device 180. In variants, selected pairs of PEs may be directly connected via a streaming interface, as in embodiments discussed later.

Interdependent load and execute operations can be more adequately handled by dual-interface systems. Therefore, in embodiments, one or more of the PEs include each, a load interface unit LD and an execution interface unit EX, see FIG. 4, where each interface unit LD, EX is configured as a dual interface system, i.e., an interface system that is connected to each of the token interconnect 120 and the control interconnect 110. The load interface unit LD can receive load commands defining data load operations from the CPU (through the control interconnect 110), as well as load tokens from connected PEs (through the token interconnect 120), which timely trigger such data load operations. Similarly, the execution interface unit EX can receive execution commands, which define operations to be performed on loaded data, as well as execution tokens, which timely trigger such operations. Tokens and commands are addressed to respective entry ports of the interface units LD, EX. E.g., a load token (respectively an execution token) is directed to an entry port of the load interface unit LD (respectively the execution interface unit EX) of the relevant PE. An example of load and execute scenario is given in the summary section for MVM operations.

1.1.3. Software endpoint component

Referring back to FIG. 1, the data processing device 10 may further comprise a software endpoint component. The software endpoint component is connected to the token interconnect 120 so as to interface the CPU 150 with each of the PEs 100 – 105 through the token

interconnect 120. The software endpoint component provides a software endpoint for the CPU. I.e., the software endpoint component is configured so that the CPU 150 can send tokens or receive tokens through the token interconnect 120.

In principle, it is not necessarily needed to configure a software endpoint in the processing device 10 – 13. However, a software endpoint is advantageous as it provides additional flexibility. I.e., it allows the CPU to produce and consume tokens, such that the CPU can further interact with the PEs over the token interconnect 120. This allows the CPU to interact with the PEs over shared resources, as discussed in the summary section. In practice, the software endpoint component is preferably integrated in the CPU. Another possibility to configure a software endpoint is to connect an additional circuit component to each of the control interconnect and the token interconnect.

1.1.4. Application to neural layers: layer-by-layer vs. depth-first executions

Referring to FIG. 3, the data processing device 10 – 13 may be configured as a neural processing device, i.e., a device that is designed to execute an ANN, or a portion thereof. At least some of the operations include MVMs and the conditions specified by the commands result in executing neural layers, or portions thereof, of the ANN. That is, the conditions of executions as specified in the commands amount to assigning executions of neural layers, or parts thereof, to the PEs 100 – 105. The correct orchestration of the tasks is achieved through sending/receiving tokens over the token interconnect 120.

In principle, the executions may be performed in a layer-by-layer fashion, see FIG. 3 (LHS). That is, initial tasks A, B, C (corresponding to neural layers in a layer-by-layer ordering, from top to bottom) may be split into operations $A_1, A_2, \dots, B_1, B_2, \dots, C_1, C_2, \dots$, which are sequentially executed according to a linear task schedule.

In variants, advantage is taken from the token-driven orchestration to execute the ANN in a depth-first manner, see FIG. 3 (RHS). One reason for doing so is that, typically, for some, if not many, of the neural layers, only a small fraction of the input data is required to compute the final tensor. Now, data locality can be exploited to reduce the data movements: Instead of executing one layer strictly after the next, layers can be partly executed in an alternating fashion. That is, the layers are alternately executed in parts. Thus, in embodiments, the conditions are specified in the commands so that they cause portions of the neural layers to be executed in a depth-first manner (each of these portions corresponds to one or more basic operations). As a result, portions of the neural layers are executed in an alternating fashion.

Note, all of the neural layers may potentially benefit from this approach. In variants, only some of the neural layers may be executed in a depth-first manner, a concept that is sometimes referred to as ‘layer fusion’.

As shown in FIG. 3, the required operations can be re-ordered in a depth-first manner, this leading to $A_1, B_1, \dots, A_2, B_2, \dots, A_3, B_2, \dots$, instead of $A_1, A_2, \dots, B_1, B_2, \dots, C_1, C_2, \dots$. The re-ordered operations can then be mapped onto the PEs, hence leading to a sequence of executions of the mapped operations as shown on the RHS. Thus, it is possible to map a linear task schedule on PEs, where portions of the neural layers execute in a depth-first manner. This reduces the memory requirements for the intermediate tensors between the alternating executed layers, as only a moving window of the tensor required to cover the function support of the next layer need to be stored.

1.1.5. Hardware acceleration devices

In principle, the PEs can be any processors, whether conventional or special-purpose processors. For example, some of the PEs can include vector processing units and/or systolic arrays. A systolic array is a network of processors that rhythmically compute and pass data through a grid of compute units to efficiently perform parallel computations, see FIG. 4. In addition, at least some of the PEs may include an in-memory compute (IMC) unit 100a, see FIG. 4. Various configurations can be contemplated, whether relying on IMC units only, systolic arrays only, conventional processors only, or a mix of processor types, as assumed in FIG. 4.

FIG. 7 shows an IMC unit 100a, which has a crossbar array structure 1500 adapted for performing MVMs. The IMC unit 100a includes input lines 1510 and output lines 1520, which are interconnected at cross-points (i.e., junctions). The crossbar array structure 1500 defines $N \times M$ cells 1540, which comprise respective memory systems 1560. Such a structure allows *in situ* computations, such that the unit 100a can be regarded as an in-memory computing device. The number of input lines 1510 and output lines 1520 will typically be on the order of several hundreds to thousands of lines. For example, arrays of 256×256 , 512×512 , or 1024×1024 , may be contemplated, although N need not necessarily be equal to M .

As illustrated in FIG. 8, each memory system 1560 is designed to store K weights, where $K \geq 2$. Each cross-point of the crossbar configuration corresponds to a cell and each cell involves a memory system 1560 capable of storing K weights. Such weights are noted $W_{i,j,k}$ in FIG. 8, where i runs from 1 to N , j from 1 to M , and k from 1 to K . For example, K may be equal to 4,

as assumed in FIG. 9. Thus, each cell can store up to K weight values. As a whole, the crossbar array structure 1500 includes $N \times M$ memory systems 1560, which can store K sets of $N \times M$ weights. So, as a whole, the cells 1540 can store $K \times N \times M$ weight values, which can be regarded as forming K bidimensional arrays, each having a dimension $N \times M$, as shown in FIG. 9. N and M correspond to the numbers of input lines 1510 and output lines 1520, respectively.

The IMC unit further includes a selection circuit, which is configured to enable $N \times M$ weights as active weights. In operation, the selection circuit selects, for each memory system 1560, a weight from its K weights and sets the selected weight as an active weight. Preferably, the selection circuit allows one of the K sets of $N \times M$ weights to be selected at a time. This amounts to selecting one array of $N \times M$ weight values at a time, this corresponding to $k = 1, 2, 3$, or 4 in FIG. 9. Thus, one of the K sets of $N \times M$ weight values can be enabled as an active set of weight values, prior to performing MVMs based on the active weight values. Each array of $N \times M$ weight values may possibly be loaded one at a time (preferably in a compressed form and decompressed at the IMC unit), in one of the K sets, while performing MVMs based on the currently active array (which corresponds to a different value of K). Only one of the K sets of weight values is active at a time: the IMC device performs MVM operations based on the weight values as stored in the currently active array.

The advantage of such an IMC unit 100a is that the weight values may be locally switched, without having to first transmit the corresponding data, i.e., without idle times. Note, new weight values may still be proactively fetched and decompressed, as background tasks, while executing MVMs based on a currently active set of weight values. For example, as illustrated in FIG. 9, the $N \times M$ cells 1540 may be designed to store four sets of $N \times M$ weight values (i.e., $K = 4$). Each dataset newly received includes $N \times M$ weight values. Thus, the received data span a whole weight set, which is meant to be stored in one of the K sets. This way, the weight values can be proactively loaded and decompressed, one array at a time, to avoid or reduce idle times.

As shown in FIG. 8, components $x_1 \dots x_N$ of input vectors x are encoded as signals applied by an input unit 1001 to the input lines of the crossbar array. This causes the crossbar array to perform MVMs by way of multiply-accumulate (MAC) operations. This leads to output signals encoding values $y_1 \dots y_M$, i.e., a vector y . Because vector components are injected in the crossbar array structure, the operations performed could be referred to as ‘vector-matrix multiplications’, instead of matrix-vector multiplications. Because repeated vector-matrix multiplication are performed, eventually such operations amount to performing matrix-matrix

multiplications. In the literature, all such operations are commonly referred to as matrix-vector multiplications.

Each memory system 1560 may for instance include serially connected memory elements, which store respective bits of the weight stored in the corresponding cell; the MAC operations are performed in a bit-serial manner in that case. The memory elements may for instance be static random-access memory (SRAM) devices. However, the crossbar array structure 1500 may, in principle, be equipped with other types of electronic memory devices (e.g., flash cells, memristive devices, etc.). Any type of memristive devices can be contemplated, such as phase-change memory cells (PCM), resistive random-access memory (RRAM), as well as electro-chemical random-access memory (ECRAM) devices.

The memory elements typically form part of a multiply-and add circuit (as suggested in FIG. 8), whereby each column includes N multiply-and add circuits, so as to efficiently perform the MAC operations. As per the crossbar configuration, M MAC operations are being performed in parallel, during each calculation cycle. Note, the operations performed at every cell correspond to two scalar operations, i.e., one multiplication and one addition. Thus, the M MAC operations imply $N \times M$ multiplications and $N \times M$ additions, meaning $2 \times N \times M$ scalar operations in total.

Output signals obtained in the M output lines 1520 are read out by a readout circuitry 1600 to obtain corresponding values $y_1 \dots y_M$. In practice, several calculation cycles need be successively performed, whereby weights are locally enabled (i.e., selected and set as active weights) at each cycle, prior to feeding components of an N -vector to perform MAC operations and read the output values. Such output values may correspond to partial values, which may advantageously be accumulated locally, e.g., at the readout circuitry 1600 or the near-memory processing device 1800. I.e., the readout operations may not only aim at extracting the output values, but also accumulating them with previous output values, if necessary. One or more near-memory processing units 1800 may actually be arranged in output of the output lines 1520 to efficiently evaluate mathematical functions (such as activation functions) applied to the neuron outputs.

An input/output (I/O) unit 1900 is used to interface the IMC unit 100a with the control interconnect and token interconnect. As noted earlier, the I/O unit may enable a load interface unit and an execution interface unit, where each of the load interface unit and the execution

interface unit is configured as a dual interface system that is connected to each of the token interconnect 120 and the control interconnect 110.

Note, while FIGS. 8 and 9 assume that the IMC device can store several sets of weights ($K > 1$), it should be kept in mind that (some of) the PEs may also be configured as simple IMC devices capable of storing only one set of weights at a time (i.e., $K = 1$), as noted in the summary section.

1.1.6. First-in-first-out channels

The transfer of data, commands, and/or tokens, is advantageously handled in a first-in-first-out (FIFO) manner, where possible. The PEs can be regarded as forming ordered pairs of PEs, inasmuch as information flowing from one element of a pair to the other likely differs from the information coming from the other element of this pair.

One or more ordered pairs of the PEs may be connected through a data streaming interface. In the example of FIG. 1, the ordered pairs (PE#3, PE#4) and (PE#4, PE#5) are directly connected through such a data streaming interface, represented by a dotted-dashed arrow. Data streaming interfaces make it possible to move data from one PE to another without having to write/load data to the local, shared memory. Such streaming interfaces will normally be designed to make it possible to synchronize access before reading the data streamed. Any such data streaming interface is preferably implemented as a FIFO channel, which automatically determines the order in which the data streamed is to be handled by the receiving PE. Note, the data streamed does not need to be orchestrated. If a PE is not ready to be produce or consume data, this PE will simply stall.

Not all of the PEs connected by data streaming interfaces need to be interfaced with the memory device 180 through the data interconnect 170. For example, in FIG. 1, PE#4 does not need to be connected to the memory device 180, since it directly receives data from the previous PE and directly streams data to the next PE. More generally, any number of ordered pairs of PEs may be connected through a data streaming interface. The precise arrangement of data streaming interfaces depends on the desired dataflow architecture. Streaming will advantageously be used where the data is meant to be produced and read in a linear manner. However, streaming interfaces are not practical for PEs meant to access (e.g., read) data according to a non-linear access pattern.

Other types of channels may possibly be devised as FIFO channels. For instance, the control interconnect may interface with each PE of the PEs 100 – 105 through a buffer designed to

queue commands dispatched by the CPU, for the commands to be processed in a FIFO manner by each PE, in operation. Likewise, the PEs and the token interconnect may be jointly configured to allow unidirectional communication channels between ordered pairs of the PEs, where each of the unidirectional communication channels behaves as a FIFO channel. Thus, tokens can be passed and processed in a FIFO manner, such that tokens do not need to encode the address of the recipient PE. Even more so, tokens can be encoded as mere bit values, for reasons that will become apparent later.

Various implementations of the token interconnect 120 can in principle be contemplated. For instance, the token interconnect may include distinct connectors, which interface respective pairs of the PEs. Preferred, however, is to design the token interconnect 120 for it to enable token channels working as a two-signal handshake interface, ensuring two-way handshakes. The two-way handshake is a communication protocol designed to ensure reliable message exchange between a sender and a receiver. It hinges on two signals: a “valid” signal from the sender, indicating a message is ready to be sent, and a “ready” signal from the receiver, indicating it can accept the message. This process ensures synchronized data transfer, with the transaction proceeding only when both signals are present. In the present context, the data processing devices 10 – 13 are preferably configured to operate the token interconnect 120 for it to enable a two-signal ready/valid handshake interface between any two connected PEs. Note, for PEs having load and execution interface units, any of the load and execution interfaces may rely on a two-signal ready/valid handshake.

Care should be taken not to exceed the capacity of the token channels. To that aim, the data processing devices 10 – 13 preferably include hardware-implemented counters 125, as illustrated in FIG. 6C. That is, a hardware counter is implemented in or for each unidirectional communication channel, where each channel connects a pair of PEs. The counters are configured to monitor their respective channels. They increment or decrement respective counter values depending on whether tokens are sent or consumed through the respective channels, see also section 2.

1.1.7. Data movement engines

In embodiments, the data processing devices 10 – 13 further include a data movement engine (DME) 160, which is connected to each of the control interconnect 110, the token interconnect 120, and the data interconnect 170. The DME enables data exchanges between the data processing device and another device, e.g., another data processing device. Since the DME 160

connects to the control interconnect 110 and the token interconnect 120, it is possible to synchronize operations between several data processing devices 10 – 13, as discussed below in reference to another aspect of the invention. Note, the DME is preferably implemented as a direct-memory access (DMA) controller (or DMA engine), as assumed in the accompanying drawings.

1.2. Data processing system (cluster of data processing devices)

Referring to FIG. 2, another aspect of the invention concerns a data processing system 1, which includes a set of data processing devices 10 – 12 such as described above. Thus, the data processing devices 10 – 12 play the role of core computing devices. In addition, the data processing system 1 includes a system-level memory device 18, a system-level control unit (e.g., a CPU 15, as assumed in the following), and a system-level data interconnect 17, which interfaces the system-level memory device 18 with each of the core computing devices 10 – 12 and the system-level CPU 15. In addition, the system 1 preferably comprises a system-level data movement engine 16, which is connected to the system-level data interconnect 17. Moreover, the system 1 may include a system-level token interconnect 19, which is connected to each of the system-level CPU 15, the system-level data movement engine 16 (if any), and each of the core computing devices 10 – 12. The system-level token interconnect is typically connected to a core computing device through its CPU 150.

The CPU of each core computing device 10 – 12 acts as a local controller. The core computing devices can for instance independently run ANNs. In variants, each core is used to run a respective part of a same ANN, which makes it possible to increase the batch size. Another possibility is to stitch the core computing devices 10 – 12 for handling very large layers, e.g., involving more than 512 nodes or pipelining layer executions in a layer-by-layer manner. Use can advantageously be made of the system-level token interconnect to orchestrate the various operations. Alternatively, the operations can be orchestrated on the system level through a usual control interconnect.

1.3. Methods

1.3.1. Main features (runtime operation)

Referring to FIGS. 10, 11, a final aspect of the invention concerns a method of executing interdependent operations. The method relies on a data processing device 10 – 13 as described above. The method revolves around operating S40 – S60 the data processing device by operating S40 the control unit (e.g., a CPU 150, as assumed in the following), for it to dispatch

commands to the PEs 100 – 105 through the control interconnect 110, and operating S50 the PEs 100 – 105 for them to send and/or receive tokens via the token interconnect 120.

As explained earlier, the commands specify conditions for executing S60 the operations in accordance with a predetermined interdependence of the operations. The commands are preferably dispatched by repeatedly backfilling the commands over the control interconnect, so that commands can be queued at the respective PEs. The PEs 100 – 105 send and/or receive tokens in accordance with the conditions as specified in the commands. The goal is to orchestrate the executions of the operations S60. Such operations cause at least some of the PEs to read from and/or write to the memory device 180 through the data interconnect 170.

The above method steps can be extended to a cluster of data processing devices 10 – 13, e.g., to a system 1 as described above.

1.3.2. Compilation

Preliminary steps may be performed prior to operating the data processing device 10 – 13, with a view to adequately preparing the CPU commands. Such steps are typically performed at a distinct computerized device, e.g., a conventional computer. These preliminary steps revolve around compiling S20 execution instructions to identify the interdependence of operations to be performed, as well as token dependencies reflecting this interdependence. Eventually, the compilation causes to generate commands in accordance with the token dependencies generated. Such commands specify conditions that reflect the token dependencies generated by the compilation. At runtime, the CPU 150 can dispatch the commands as obtained through the compilation. The execution instructions are preferably compiled under the constraint that the CPU is to be operated according to a linear execution schedule; the CPU is subsequently operated S40 according to this linear execution schedule.

1.3.3. Applications to ANNs

As said, at least some of the operations may include MVMs, and one or more of the PEs may include an IMC unit 100a (FIG. 7), which has a crossbar array structure 1500. Such an IMC unit is well suited for executing ANN inferences. Thus, in embodiments, the data processing device 10 – 13 is operated to execute an ANN. In this case, the conditions specified by the commands cause the executions to result in executing neural layers, or portions thereof, of the ANN.

As discussed earlier in reference to FIG. 3, the data processing device may advantageously be operated to execute the neural layer parts in a depth-first manner, in accordance with the

commands sent to respective PEs 100 – 105. Each of the neural layer parts corresponds to one or more of the operations, whereby the neural layer parts are orchestrated thanks to the tokens sent and received by the PEs 100 – 105 via the token interconnect 120. As a result, portions of the neural layers are executed in an alternating fashion.

1.3.4. Counters

Referring to FIG. 11, the token interconnect is preferably configured to enable unidirectional communication channels between ordered pairs of the PEs. Such channels preferably work as FIFO channels. The method further comprises, for each channel of the unidirectional communication channels: incrementing S52 (respectively decrementing S52) a counter value each time a token is sent (respectively consumed) through each channel. In addition, the method further comprises monitoring S54, S56 the counter value against a threshold value, which is related to a token capacity of each channel. This makes it possible to detect deadlock or near-deadlock situations. For example, if the counter value reaches a threshold value, then a warning can be triggered S58 to detect a potential deadlock risk, whereby preventive measures can be taken to prevent a deadlock. The maximal capacity determine the maximal number of outstanding dependency relations. A deadlock can otherwise be prevented by enforcing compilations that are compatible with the maximal channel capacities.

1.3.5. Preferred flows (FIGS. 10 and 11)

FIG. 10 shows a preferred flow, in which a new job specification is received at step S10. At step S20, instructions for executing operations are compiled to identify a desired interdependence between these operations, determine corresponding token dependencies, and generate commands in accordance with these token dependencies. The CPU can then be operated S40 to dispatch the generated commands to the PEs through the control interconnect. The CPU may repeatedly backfill commands over the control interconnect. At step S50, the PEs are operated according to the commands, which causes the PEs to send/receive tokens via the token interconnect. This, in turn, causes the counters to increment/decrement counter values, see steps S52 – S58 explained below. Operating S50 the PEs further causes the PEs to execute S60 operations as specified in the commands, which causes the PEs to read from/write to the memory device or stream data to one another. The tokens sent and received through the token interconnect ensure a correct orchestration of the executions of operations. Notwithstanding the flow depicted in FIG. 10, it should be understood that steps S40 – S60 are largely intermingled. I.e., steps S40, S50, and S60, do actually not execute one after the other.

Rather, steps S40 – S60 execute in an interlaced manner and continue until a final result is achieved. This result is stored at step S70.

As shown in FIG. 11, a counter value is updated S52 (i.e., incremented/decremented) for each token channel upon detecting a token transmission/reception. For each token channel, the counter value is monitored S54 against a maximal channel capacity. If the counter value exceeds a threshold value indicating that the maximal channel capacity has been reached, an error message is generated S58, indicating a deadlock risk. In such a case, it is still possible to fall back on a CPU-driven synchronization, at least for some of the operation interdependences, see section 2. In practice, however, such a situation can be avoided by provisioning sufficiently large channel capacities. I.e., the capacity of any channel should exceed the sum of commands to be queued at the PEs meant to send tokens on that channel.

1.3.6. Data structure of the commands

The token interconnect 120 preferably interfaces each PE of the PEs with μ components (i.e., PEs as well as, possibly, the CPU) of the data processing device 10 – 13 through μ unidirectional communication channels working as first-in-first-out channels. In principle, the μ connected components include at least one PE. I.e., each PE connects to at least another one of the PEs. Preferably, each PE connects to each of the remaining PEs, in each direction, as in the example of FIG. 6C. As there are only three PEs in this example, each PE connects to two PEs. Since the token channels are unidirectional, this means that four token channels are needed for each PE, i.e., two channel for receiving tokens and two channels for sending tokens. More generally, each PE may connect to up to $\nu - 1$ other PEs (assuming there are ν PEs in total in a core), which means $\nu - 1$ channels for reception and $\nu - 1$ channels for transmission.

Assume for now that each PE is interfaced with μ components of the data processing device 10 – 13, where $\mu \geq 1$. In that case, the commands may advantageously capture the token dependencies thanks to 2μ bit fields. That is, each command includes μ bit fields for reception, i.e., determining up to μ tokens to be respectively received by the given element from the μ components before executing the operation, as well as μ bit fields determining up to μ tokens to be respectively sent by the given element to the μ components upon completion of the operation. In addition, each command include data determining the operation to be performed by the target PE.

As noted above, μ may be equal to $\nu - 1$, where ν corresponds to the total number of PEs that are interfaced with one another through the token interconnect 120. I.e., the token interconnect 120 interfaces each PE with each of the $\nu - 1$ remaining PEs. However, μ may also be equal to $\nu - 1 + 1$ (i.e., $\mu = \nu$), should the data processing device further comprise a software endpoint component. As explained earlier, the software endpoint component is preferably integrated in the CPU 150. The software endpoint component is connected to the token interconnect 120 so as to interface the CPU 150 with each of the PEs 100 – 105 through the token interconnect 120.

The following provides a practical illustration of the token operation. FIG. 6B shows a dependency graph 30 with six operations/nodes (A – F) running on a core with three PEs (PE#0 – PE#2). Note, these nodes A – F are unrelated to the units A – F shown in FIG. 5. Each node is executed on one PE, as annotated in the graph of FIG. 6B. The dependency between nodes is illustrated with arrows.

FIG. 6C shows the three PEs 100 – 102 of FIG. 6A, which are interconnected through the token interconnect 120. Below the three PEs is an illustration of their corresponding command FIFOs filled with the commands required to execute the graph (the top-most operation in the queue is executed first). The command format for each PE is shown further below: Each command format contains the token information (RX/TX) and the information to execute (Op). Here, the operation information (Op) just shows the letter identifying the corresponding node in the dependency graph of FIG. 6B. The token information annotates which tokens have to be received (RX) and which tokens have to be sent after the command's execution is completed. RX/TX are bitfields corresponding to the other two PEs (e.g., #1 refers to PE#1). If a RX bit is set to '1' the corresponding command operation is stalled until the target PE has received one token on the token line coming from the PE as defined in the command format definition. If there are multiple bits set to '1', this execution waits until one token has been received on all marked channels. If a TX bit is set to '1', the PE sends a token on the line corresponding to this bitfield after the operation addressed in the corresponding command was fully executed on the PE.

Various patterns (full, dotted, dashed, dotted dashed, thick lines) are used in FIGS. 6B and 6C to map the dependency arrows in the dependency graph onto the corresponding token channels. The same patterns are also used in the commands to show which bit field it relates to.

Note, the dependency indicated by a thick arrow in FIG. 6C does not need to be captured by the token mechanism; it is implicit from the order in the queue as the corresponding operations are

executed on the same PE in the correct order as guaranteed by the FIFO. In addition, after the execution of operation A on PE#0, the operation of B and C on PE#0 and PE#1 can be started and executed concurrently, while operation D on PE#2 will wait until both operation B and C are completed. All this happens without interaction with the control unit. Finally, the command FIFO guarantees a certain order in the execution, which allows dependencies between the same PEs (PE#0 and PE#1 for A to C and E to F) to be resolved by the same token channel.

The above examples assume that each PE can receive tokens from one or more other PEs and transmit tokens to one or more other PEs. The command structure can be further adapted to cases where the PEs include separate load and execution interface units.

The above embodiments have been succinctly described in reference to the accompanying drawings and may accommodate a number of variants. Several combinations of the above features may be contemplated. Examples are given in the next section.

2. Specific embodiments

The following describes techniques to efficiently tackle computations of interdependent operations, which can advantageously be applied to neural network inferences.

2.1. Preliminary remarks

For some (if not many) neural network layers, only a small fraction of the input data is typically required to compute the resulting tensor. This holds true for certain types of layers, such as convolutional layers in convolutional neural networks, as opposed to fully connected (dense) layers. This data locality can be exploited to reduce the data movements: Instead of executing one layer strictly after the next, layers can be partly executed in an alternating fashion. This reduces the memory requirements for the intermediate tensors between the alternating executed layers, as only a moving window of the tensor required to cover the function support of the next layer needs to be stored. An advantage of this approach is that partial tensors may happen to fit entirely in the lower-level caches/buffers and thus prevent data movements on the system level.

The problem of which layers of a compute graph should be executed in this ‘depth-first’ manner, and with which granularity the execution of the layer should be switched, is an optimization problem that requires carefully balancing partial tensor memory size with layer

switch overheads. The more layers are executed in this manner, the more instances of partial intermediate tensor must be stored close to the compute units. If the layer execution is switched more often, the partial intermediate tensors become smaller, at the cost of losing compute performance for the context switch in the compute unit and a more complex data dependency management. Another factor to consider is that the neural layers require a substantial number of parameters (starting with weights) for their execution. Context switching implies that a change of those parameters results in further data traffic.

Usually, the execution model of ANN inference compute graph is purely a software question, which is independent of the underlying hardware architecture. Both compute models, i.e., layer-by-layer and depth-first (also called layer fusion where only some of the layers are alternately executed in parts), can in principle be deployed on conventional compute architectures such as CPUs and GPUs. However, the efficacy, especially for the depth-first approach, can be substantially improved by optimizing the compute architecture.

A dataflow architecture (i.e., a dataflow-based computer architecture) departs from traditional von-Neuman architectures, where the execution of a task is determined by the availability of input data. Thus, one understands that dataflow architectures are suited for the execution of workloads in a 'depth-first' manner. They also enable multiple compute units to collaboratively work in parallel on a workload to avoid intermediate results being stored in global memory. However, such architectures are difficult to program and control in practice. Compilation and online data-dependency resolution is non-trivial. Compiler complexity and implementation effort are already a challenge for custom AI accelerators; they make pure dataflow architecture prohibitive for use on conventional platforms.

Hybrid dataflow architectures combine the benefits of ease of use and compiler complexity of von Neuman architectures with improved compute throughput and data-reuse of dataflow architectures. Known hybrid dataflow architectures for AI inference follow an approach where different layers are spatially distributed over multiple compute entities to avoid the overhead due to context and parameter (weights) switching. This typically results in workload balancing issues as the compute and data requirements for the various neural layers in the compute graph can be vastly different.

Modern ANN compute graphs are built by a large and diverse set of different layer flavours. While in the past only a small set of limited layer flavours were used (e.g., convolutional layers, fully connected layers, and a few simple activation function layers), today's ANNs involve

many different layer flavours to allow improved prediction performance for a smaller set of function parameters and reduced compute complexity. While the goal of early hardware accelerators for AI inference was to simply accelerate executions of a few neural layers and functions, it is now desired to provide hardware acceleration support for a diverse set of operations and anticipate future ANN layer architectures.

Conventional homogeneous architectures involving one dedicated hardware accelerator typically are no longer satisfactory enough to provide decent performance over an entire workload. Moreover, AI applications workloads typically not only consist of the core neural network but further include operations to be performed before, after, or between, neural network compute loads. For computer vision applications, for example, this implies that an architecture not only needs to provide sufficient performance for the neural network inference part but also the traditional computer vision task the application comprises. As a single compute entity can no longer optimally cover the various workloads, one may contemplate integrating different kinds of compute units (i.e., processing units) into a heterogeneous architecture and stitch together the different compute units on the system level for various operations. This loose integration prohibits units to closely collaborate.

2.2. Particularly preferred embodiments

The following relates to a scalable hybrid heterogeneous dataflow architecture, which is optimized for neural network inference. This architecture is built out of data processing devices (referred to as cores in the following), see FIG. 1, where each core contains PEs (PEs) 100 – 105 of the same or different type such as IMC units, systolic arrays, vector processors, which can all access a local, shared multi-bank memory over a high-bandwidth interconnect. In addition, the core 10 may include a DME (e.g., a DMA controller), having a similar connection configuration. The PEs are controlled by a CPU 150, which dispatches high-level commands to the PEs over a control interconnect. The commands are marked with tokens, which determine the synchronization of operations throughout the PEs. All PEs can operate independently and concurrently on different tasks, subject to timing dictated by the tokens. To reduce control program load for the CPU, execution dependencies and data hazards between the PEs can be resolved by the token network 120 connecting all PEs and the DME. The token path is physically distinct from the control and data paths, meaning that the token network interconnect 120 is distinct from the control and data interconnects. Selected pairs of the PEs can optionally be connected by a data streaming interface to move data from one PE to another without having to write/load data to the shared memory. In more sophisticated layouts, a PE 101b may include

several successive pairs of processing units A – F that are connected by a data streaming interface to match a specific compute flow, as in the data processing device 12 shown in FIG. 5.

The proposed architecture is scalable, as cores can be used as PEs in a higher hierarchical level to form a system 1, see FIG. 2. Multiple cores can possibly be connected to a system-level shared memory and controlled by a system-level CPU. What is more, systems 1 may possibly be clustered, too. That is, the scalability allowed by the present approach is both recursive and hierarchical.

To allow heterogeneous PEs to be integrated in the same core, different memory interfaces towards the local shared memory are supported. PEs can have one or multiple separate read and write interfaces towards the memory to support streaming accelerators, which read data over one or multiple interfaces and write results over one or multiple interfaces.

All PEs implement a common command-based control interface, which allows the CPU to dispatch tasks to the PEs. This interface is preferably implemented as a queue (e.g., a FIFO) to allow multiple outstanding commands to queue in the PEs. The command interfaces in all PE's are connected over the token network to resolve execution dependencies between the PEs in hardware. Note, this token interconnect 120 (or token network) should be distinguished from a token ring, even though tokens are passed around several nodes in both cases (the purpose and architecture, however, differ).

Execution dependencies (which are conditioned on data dependencies) are resolved during compile time and annotated in the metadata of the commands sent to the PEs. A PE can be set to wait (in accordance with a command) for one or more tokens to be received by another unit before (and in order to) be able to start executing a task (e.g., in accordance with that command) and/or set to send a token to another unit once a command is fully processed.

This makes it possible to guard dependencies (e.g., read after write) to prevent a PE from starting loading data before another PE tasked with creating this data has written it. In that case, the command sent to the data-producing (write) PE is annotated to send a token to the data consuming (read) PE. The command sent to the data consuming (read) PE is annotated not to start executing (i.e., reading data) before a token has been received from the data producing (write) PE. In the same way, anti-dependencies (write after read) can also be guarded, i.e., preventing a PE from overwriting data in memory with new data before it has been consumed.

The token network provides a connection for any two PEs that are accessing a shared resource. Token channels are implemented as FIFO channels. The latter are designed with sufficient capacity, which can be bounded by a function of the size of command FIFOs of the PEs that the token channels connect. The token channel is easy to fabricate (e.g., as a two-signal interface) and allows much faster synchronization since the token path is not encumbered by control traffic.

For additional flexibility, the system further comprises a software token endpoint, which allows the CPU 150 to produce or consume (i.e., wait for) tokens, such that the CPU can interact with the PEs over shared resources.

For example, the CPU can already backfill commands into the PE's command FIFO, before the CPU has completed its access to a shared resource. This is useful when the PE requires control information (besides data) to be accessed from shared resources, e.g., a DMA descriptor or program instructions, which are written by the CPU. Once the data is confirmed to be written to the destination, the CPU can clear the command in the PE's FIFO for execution by sending a token to that PE over the software endpoint.

Similarly, the CPU can obtain information on which commands have been executed, by marking commands to send tokens to the software endpoint. This provides an efficient way to implement barriers in the software control code and cause the CPU to wait until certain commands have been fully executed in the PEs, without having to poll status information from the PEs.

As said, selected pairs of PEs can be connected by one or multiple unidirectional data streaming interfaces. This makes it possible to move data from one PE to another without writing it to the shared memory, synchronize the access, and then have the other PE read it. PEs benefitting from data streaming interfaces are normally disconnected from the shared memory. The streaming interfaces can for instance be implemented as data/valid/ready streams to allow bidirectional handshaking. This allows the downstream (sink) PE to wait for results from the source PE and allows the downstream (sink) PE to backpressure the source PE if it is not ready to consume the incoming data yet.

Any command that is sent to a PE can optionally trigger this PE to consume or produce data from/to its streaming interfaces. A command will trigger a PE to produce/consume a given number of data transactions over the streaming interfaces. Typically, when a source PE is instructed to produce L data elements with a single command, also the sink PE will be

instructed to consume L data elements. However, it is also possible that multiple commands are sent to one of the PEs to produce/consume partial data multiple times, while the other PE receives only one command for all the data.

Additionally, the data streaming interfaces can be implemented as FIFO channels. This provides additional buffer capability to the system, which may for example enable data-rate balance between PEs or allow a source PE to already produce data while a sink PE is still busy with a previous task.

Moreover, those streaming interfaces provide a means to tightly connect very large PEs, which require a lot of silicon area and may not be closely placed, as pipeline registers can be inserted in the streaming interface to bridge far distances without impairing timing (the maximal frequency the system can run).

Programming model. To ease the programmability, the CPU preferably follows a linear execution schedule irrespective of whether a compute graph is executed in a ‘layer-by-layer’ or ‘depth-first’ manner.

In conventional (prior) approaches, the CPU dispatches a task to a PE and waits until the PE reports back that the task is fully executed, meaning all results of the task have been confirmed to be written to their target location. Then a dependent task is dispatched. In practice, this works by dispatching a command into the PE’s command FIFO, marking the command to send a software token to the CPU, and then wait for the software token from that PE unit to arrive at the software endpoint of the token network with a token synchronization barrier. Once this token has arrived, the procedure is repeated for the next task. This default mode is not very efficient as it does not allow parallel processing and has a high control overhead as the PE is idling between tasks while the CPU is running the management routines for task dispatching. However, it guarantees that no execution dependencies are violated as all PEs come to a full stop after each task when accesses to a shared resource are completed.

In the present context, this default mode may be used as a starting point for the compiler. In embodiments, this default mode may further be used as a fallback mode, for which workload-compilation is always possible.

Starting from the default mode, the compiler can start overlapping tasks by removing the synchronization barriers and model execution dependencies by inserting token dependencies. This can for example mean that:

- One PE reports to another PE when data is written and ready to be read;

- One PE waits with overwriting data until another PE has confirmed that data was fully read; or
- Any two PEs otherwise communicate by way of tokens to solve any conflict over a shared resource.

Assuming all dependencies can be removed that way, all barriers in the linear execution schedule of the CPU can be removed. At this point, the CPU is constantly backfilling commands in the command FIFOs of the PEs, while the underlying operations are being executed on the PEs in accordance with the tokens. Since the command FIFOs have limited capacity, the CPUs command backfilling routing can be stalled if the PE's command FIFOs are full.

When the workload is done, the last PE executing the last task of the workload writing the final results will have to inform the CPU by sending a software token. Note that the CPU could have already started filling the commands for the next workload at this point.

As said, preferred implementations make it possible to fall back to the default mode to guarantee that compilation is always possible with reasonable complexity. Trade-offs are possible, which make it possible to optimize programming complexity vs. efficiency at runtime.

Streaming interfaces. If a PE receives a command to produce data for a streaming interface, it can present the data at the streaming interfaces when available. If the downstream (sink) PE is not ready, or the FIFO in the streaming interface (if any) is full, the unit will back stall. If a PE receives a command to read data from a stream interface it will stall until data is ready to be consumed. Data will flow only if all PEs connected over a streaming path are executing a command accessing the interface.

This programming model, as enabled by the underlying architecture, gives rise to an understandable programming model, which enables multiple PEs to operate concurrently to resolve dependencies with a hardware token mechanism, instead of relying on CPU interactions. It also makes it possible to queue multiple commands per PE to enable back-to-back executions of tasks without overhead and without requiring CPU interactions for synchronization.

Still, the above approach comes with additional compiler complexity: the compiler needs to model the dependencies and annotate this correctly in the generated task schedule and commands. This has the inherent risk of making the compiler complexity infeasible for

workloads that have very complicated data interdependencies. However, since the architecture allows to fall back on the simple default mode, one can reduce the compiler complexity to a manageable degree and only use the token network when the achieved performance gains justify the additional compiler complexity. On the other hand, giving the compiler the possibility to model and deploy the dependency resolution policy opens the door to substantial optimization and hardware complexity reduction, as one can resolve dependencies on high-level data structures (e.g., allocated buffers or entire data tensors placed in memory), as opposed to a more fine-grained, data-element wise data-dependency resolution.

The proposed architecture also inherently enables ‘depth-first’ processing. Starting from a linear layer-by-layer schedule, tasks reflecting entire compute graph nodes can be split up and reordered in the schedule for alternating execution (FIG. 3). As explained earlier, this reduces the size of required buffers in the memory of intermediate results. Different tasks can be executed on the same PE or different PE units depending on which PE unit provides the best compute performance for a given operation. In the default execution mode, tasks are executed sequentially over the entire core, with the CPU synchronizing with the PE after every task (FIG. 3). The compiler can now remove synchronization barriers from the schedule and insert token dependencies into the commands. A PE can now start its execution as soon as the dependencies are fulfilled and without the need for any CPU interaction/synchronization. The tasks are mapped to different PEs, which allows the linear task schedule to be executed in parallel on the various PEs. This principle is similar to out-of-order execution in modern CPU, except that it is happening on PEs with large-scale tasks as opposed to individual instructions, and that dependencies are resolved by the compiler and not by the hardware.

The example shown in FIGS. 6A – 6C assumes a token network with three PEs. Each token network participant (PEs, including a software endpoint implemented at one of the PEs, in the token network, or the CPU) has a number of token consume ports and token produce ports. If the token network is fully connected, then each PE has $\nu - 1$ consume ports and $\nu - 1$ produce ports. In the example above, which involves $\nu = 3$ PEs, the number of consume ports and produce ports on each PE is equal to 2. In variants, the network can also be partially connected. Two PEs need to be connected only if they are competing over a shared resource.

For example, the PE 100a in FIG. 4 is an IMC unit. Only the LD and EX interfaces have access to the IMC memory space (this space is only accessible by the LD and EX interfaces). However, this IMC memory space is not accessible by PE 101a (a systolic array in this example).

A token channel has a certain capacity Q to store a number of tokens in transit. A producer PE can insert multiple tokens into the channel, which can then be consumed one by one by a consumer PE. For the system to be guaranteed deadlock free, it should be ensured that the capacity of each token channel is never exceeded. If this would happen then a PE would no longer be able to complete a task, as it would no longer be able to produce a token when completed; this can potentially deadlock the system. In practice, this issue can be solved by provisioning sufficient token channel capacities. As the tokens contain no payload, a FIFO with a capacity Q does not have to be implemented with Q memory elements. It can instead be realized as a hardware counter, see FIG. 6C, representing the fill state of the FIFO. This only requires approximately $\log_2(Q)$ memory elements. The counter is increased if a token is inserted in the channel, and decreased if a token is removed. If the counter reaches a maximal value, then an error condition can be triggered S58 to signal a potential deadlock risk (the FIFO is reaching an overflow condition). The capacity Q limits how many outstanding dependency relations between two PEs can be afforded at any given point in time.

3. Technical implementation details

The present devices 10 – 13 and systems 1 may form part of a larger computer system, involving a server, which interacts with clients, who may be natural persons (interacting via personal computers), processes, or machines. Client requests are managed by the server, which may notably be configured to map a given computing task onto vectors and weights, which are then passed to PEs of the devices 10 – 13 or systems 1. The overall computer system may for instance be configured as a composable disaggregated infrastructure, which may further include other hardware acceleration devices, e.g., application-specific integrated circuits (ASICs) and/or field-programmable gate arrays (FPGAs). Of course, many other architectures can be contemplated. For example, the system 1 may be configured as a standalone system or as a computerized system connected to one or more general-purpose computers. The system 1 may notably be used in a distributed computing system, such as an edge computing system.

Computerized devices and systems as described herein can be designed for implementing embodiments of the present invention as described herein, starting with the methods described in sections 1 and 2. In that respect, it can be appreciated that such methods are essentially non-interactive, i.e., automated. Automated parts of such methods will typically be implemented as a combination of hardware and software.

Aspects of the present invention are described herein notably with reference to a flowchart and block diagrams. It will be understood that each block, or combinations of blocks, of the flowchart and the block diagrams can be implemented through computer readable program instructions. The flowchart and the block diagram in the accompanying drawings illustrate the architecture, functionality, and operation of possible implementations of the data processing devices and systems, and methods of operating them, according to various embodiments of the present invention.

While the present invention has been described with reference to a limited number of embodiments, variants, and the accompanying drawings, it will be understood by those skilled in the art that various changes may be made, and equivalents may be substituted without departing from the scope of the present invention. In particular, a feature (device-like or method-like) recited in a given embodiment, variant or shown in a drawing may be combined with or replace another feature in another embodiment, variant, or drawing, without departing from the scope of the present invention. Various combinations of the features described in respect of any of the above embodiments or variants may accordingly be contemplated, that remain within the scope of the appended claims. In addition, many minor modifications may be made to adapt a particular situation or material to the teachings of the present invention without departing from its scope. Therefore, it is intended that the present invention is not limited to the particular embodiments disclosed, but that the present invention will include all embodiments falling within the scope of the appended claims. In addition, many other variants than explicitly touched above can be contemplated. For example, other types of memory elements and processing elements can be contemplated.

CLAIMS

1. A data processing device (10 – 13) for executing interdependent operations, the device comprising:

- a control unit (150);
- a memory device (180);
- processing elements (100 – 105), which are configured to execute operations; and
- distinct interconnects (110, 120, 170), which include
 - a data interconnect (170) interfacing the memory device (180) with at least some of the processing elements (100 – 103, 105),
 - a control interconnect (110) interfacing the control unit (150) with each of the processing elements (100 – 105); and
 - a token interconnect (120) interfacing each of the processing elements with at least another one of the processing elements (100 – 105),

wherein

the control unit (150) is configured to dispatch commands to the processing elements (100 – 105) through the control interconnect (110), the commands specifying conditions for executing the operations in accordance with a predetermined interdependence of said operations, and

the processing elements (100 – 105) are configured to send and/or to receive tokens via the token interconnect (120), in accordance with said conditions, to orchestrate said executions and accordingly cause at least some of the processing elements to read from and/or write to the memory device (180) through the data interconnect (170), in operation.

2. The data processing device (10, 11) according to claim 1, wherein:

the processing elements (100 – 105) include a first processing element (100) and a second processing element (101), which are configured to execute a first operation and a second operation of said operations, in accordance with a first command and a second command of said commands, respectively; and

the second command is designed so that an execution of the second operation by the second processing element (101) is conditional on receipt by the second processing element of an execution token transmitted by the first processing element (100) via the token interconnect (120) upon completing the first operation.

3. The data processing device (10, 11) according to claim 2, wherein
the first operation is designed to cause the first processing element to write first data to the memory device (180) through the data interconnect (170), and
the second operation is a data load operation, designed to cause the second processing element to read at least part of the first data from the memory device (180) through the data interconnect (170).
4. The data processing device (11) according to any one of claim 1 to 3, wherein
one or more of the processing elements (100a, 101a) include each, a load interface unit and an execution interface unit, and
each of the load interface unit and the execution interface unit is configured as a dual interface system that is connected to each of the token interconnect (120) and the control interconnect (110).
5. The data processing device (10 – 13) according to any one of claim 1 to 4, wherein
the data processing device further comprises a software endpoint component, which is connected to the token interconnect (120) so as to interface the control unit (150) with each of the processing elements (100 – 105) through the token interconnect (120), and
the software endpoint component is preferably integrated in the control unit.
6. The data processing device (10 – 13) according to any one of claim 1 to 5, wherein
at least some of said operations include matrix-vector multiplications and said conditions cause said executions to result in executing neural layers, or portions thereof, of an artificial neural network, whereby the data processing device (10) is configured to execute the artificial neural network.
7. The data processing device according to claim 6, wherein
said conditions are specified in the commands such that they cause portions of the neural layers to be executed in a depth-first manner, each of the portions corresponding to one or more of said operations, whereby the neural layers are partially executed in an alternating fashion, in operation.
8. The data processing device (10 – 13) according to any one of claim 1 to 7, wherein

at least some of said operations include matrix-vector multiplications, or MVMs,
at least one of said at least some of the processing elements includes an in-memory compute unit (100a), or IMC unit, which has a crossbar array structure (1500) adapted for performing such MVMs, and

the crossbar array structure (1500) preferably includes

$N \times M$ cells (1540) comprising respective memory systems (1560), each designed to store K weights, $K \geq 2$, whereby the crossbar array structure (1500) includes $N \times M$ memory systems (1560) adapted to store K sets of $N \times M$ weights, and

a selection circuit configured to enable $N \times M$ weights as active weights by selecting, for each of the memory systems (1560), a weight from its K weights and setting the selected weight as an active weight.

9. The data processing device (11) according to claim **8**, wherein at least one (101a) of the processing elements (100 – 105) is a systolic array or a vector processing unit.

10. The data processing device (10) according to any one of claims **1** to **9**, wherein

one or more ordered pairs of the processing elements are connected through a data streaming interface,

the data streaming interface is preferably implemented as a first-in-first-out channel, and

at least some (104) of the processing elements of the one or more ordered pairs are preferably not interfaced with the memory device (180) through the data interconnect (170).

11. The data processing device (10) according to any one of claims **1** to **10**, wherein

the processing elements and the token interconnect are jointly configured to allow unidirectional communication channels between ordered pairs of the processing elements, each of the unidirectional communication channels behaving as a first-in-first-out channel.

12. The data processing device (10) according to claim **11**, wherein

the device further includes hardware-implemented counters, which are configured to monitor respective channels of the unidirectional communication channels, and increment or decrement respective counter values depending on whether tokens are sent or consumed through the respective channels.

- 13.** The data processing device (10) according to any of claims **1** to **12**, wherein
the data processing device (10) is configured to operate the token interconnect (120)
for the latter to enable a two-signal ready/valid handshake interface with each of the processing
elements.
- 14.** The data processing device (10) according to any one of claims **1** to **13**, wherein
the data processing device (10) is designed as an integrated circuit, and
the memory device (180), the control unit (150), and the processing elements (100 –
105), are all co-integrated in this integrated circuit.
- 15.** The data processing device (10) according to any one of claims **1** to **14**, wherein
the control interconnect interfaces with each element of the processing elements (100
– 105) through a buffer designed to queue commands as sent by the control unit, for said
commands to be processed in a first-in-first-out manner by said each element, in operation.
- 16.** The data processing device (10) according to any one of claims **1** to **15**, wherein
the data processing device (10) further includes a data movement engine (160), which
is connected to each of the control interconnect (110), the token interconnect (120), and the
data interconnect (170).
- 17.** A data processing system comprising:
a set of data processing devices (10 – 12), each according to the data processing device
(10) according to any one of claims **1** to **16**;
a system-level memory device (18);
a system-level control unit (15); and
a system-level data interconnect (17) interfacing the system-level memory device (18)
with each of the data processing devices (10 – 12) and the system-level control unit
(15),
wherein
the system preferably comprises a system-level data movement engine (16), which is
connected to the system-level data interconnect (17), and
the system preferably comprises a system-level token interconnect (19), which is
connected to each of
the system-level control unit (15),

the system-level data movement engine (16), if any, and each of the data processing devices (10 – 12).

18. A method of executing interdependent operations, wherein the method comprises:

providing a data processing device (10), which includes a control unit (150), a memory device (180), processing elements (100 – 105) configured to execute operations, and distinct interconnects (110, 120, 170) including a data interconnect (170) interfacing the memory device (180) with at least some of the processing elements (100 – 105), a control interconnect (110) interfacing the control unit (150) with each of the processing elements, and a token interconnect (120) interfacing each of the processing elements with at least another one of the processing elements; and

operating (S40 – S60) the data processing device (10) by

operating (S40) the control unit (150) to dispatch commands to the processing elements (100 – 105) through the control interconnect (110), the commands specifying conditions for executing the operations in accordance with a predetermined interdependence of said operations, and

operating (S50) the processing elements (100 – 105) to send and/or receive tokens via the token interconnect (120), in accordance with said conditions, to orchestrate said executions and accordingly cause at least some of the processing elements to read from and/or write to the memory device (180) through the data interconnect (170).

19. The method according to claim **18**, wherein the control unit (150) is operated (S40) so that said commands are dispatched by repeatedly backfilling said commands over the token interconnect.

20. The method according to claim **18** or **19**, wherein

at least some of said operations include matrix-vector multiplications, and

at least one (101) of said at least some of the processing elements (100 – 105) includes an in-memory compute unit, which has a crossbar array structure (1500).

21. The method according to claim **18**, **19**, or **20**, wherein

the token interconnect enables unidirectional communication channels between ordered pairs of the processing elements, wherein the unidirectional communication channels preferably work as first-in-first-out channels, and

the method further comprises, for each channel of the unidirectional communication channels:

incrementing (S52), respectively decrementing (S52), a counter value each time a token is sent, respectively consumed, through said each channel; and

monitoring (S54, S56) the counter value against a threshold value, which is related to a token capacity of said each channel.

22. The method according to any one of claims **18** to **21**, wherein the method further comprises, prior to operating the data processing device (10),

compiling (S20) execution instructions to determine token dependencies reflecting said interdependence and generate said commands in accordance with the token dependencies determined, whereby the conditions specified by the commands reflect the identified token dependencies.

23. The method according to claim **22**, wherein

the execution instructions are compiled under a constraint that the control unit is to be operated according to a linear execution schedule, and

the control unit is operated according to said linear execution schedule.

24. The method according to any one of claims **18** to **23**, wherein

the data processing device (10) is operated to execute an artificial neural network,

at least some of said operations include matrix-vector multiplications, and

said conditions cause said executions to result in executing neural layers, or portions thereof, of the artificial neural network.

25. The method according to claim **24**, wherein

the data processing device (10) is operated to execute said neural layer parts in a depth-first manner in accordance with the commands sent to said respective ones of the processing elements (100 – 105),

each of the neural layer parts corresponds to one or more of said operations, whereby the neural layer parts are orchestrated thanks to the tokens sent and received by the processing

elements (100 – 105) via the token interconnect (120), such that the neural layers are partially executed in an alternating fashion.

26. The method according to any one of claims **18** to **25**, wherein

the token interconnect (120) interfaces each processing element of the processing elements with μ components of the data processing device (10), the μ components including at least another one of the processing elements, and

each of the dispatched commands includes:

data determining an operation to be performed by a given element of the processing elements;

μ bit fields determining up to μ tokens to be respectively received by the given element from the μ components before executing said operation; and

μ bit fields determining up to μ tokens to be respectively sent by the given element to the μ components upon completion of said operation.

27. The method according to claim **26**, wherein

$\mu \geq \nu - 1$, where ν corresponds to the total number of processing elements that are interfaced with one another through the token interconnect (120), whereby the token interconnect (120) interfaces each processing element of the processing elements with each of the $\nu - 1$ other processing elements of said processing elements.

28. The method according to claim **27**, wherein

$\mu = \nu$, and

the data processing device further comprises a software endpoint component, which is integrated in the control unit (150) and is connected to the token interconnect (120) so as to interface the control unit (150) with each of the processing elements (100 – 105) through the token interconnect (120).