US012386602B2

(12) **United States Patent**
Wang et al.

(10) **Patent No.: US 12,386,602 B2**
(45) **Date of Patent: Aug. 12, 2025**

(54) **OPERATION FUSION IN NESTED META-PIPELINE LOOPS**

(71) Applicant: **SambaNova Systems, Inc.**, Palo Alto, CA (US)

(72) Inventors: **Fei Wang**, Palo Alto, CA (US); **Weihang Fan**, Mountain View, CA (US); **David Alan Koeplinger**, Egg Harbor, NJ (US)

(73) Assignee: **SambaNova Systems, Inc.**, Palo Alto, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 169 days.

(21) Appl. No.: **18/130,642**

(22) Filed: **Apr. 4, 2023**

(65) **Prior Publication Data**

US 2023/0315411 A1 Oct. 5, 2023

Related U.S. Application Data

(60) Provisional application No. 63/327,270, filed on Apr. 4, 2022.

(51) **Int. Cl.**
G06F 8/41 (2018.01)

(52) **U.S. Cl.**
CPC ***G06F 8/4452*** (2013.01); ***G06F 8/433*** (2013.01); ***G06F 8/452*** (2013.01)

(58) **Field of Classification Search**
CPC G06F 8/4452; G06F 8/433; G06F 8/452
USPC 717/120–178
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

10,445,451	B2 *	10/2019	Fleming	G06F 12/0802
10,452,452	B2 *	10/2019	Hetzel	G06F 9/5055

(Continued)

FOREIGN PATENT DOCUMENTS

WO	2010142987 A1	12/2010

OTHER PUBLICATIONS

Seffrin, Andre, and Sorin A. Huss. "Ensuring secure information flow in partially reconfigurable architectures by means of process algebra analysis." 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications. IEEE, 2011. pp. 443-450. (Year: 2011).*

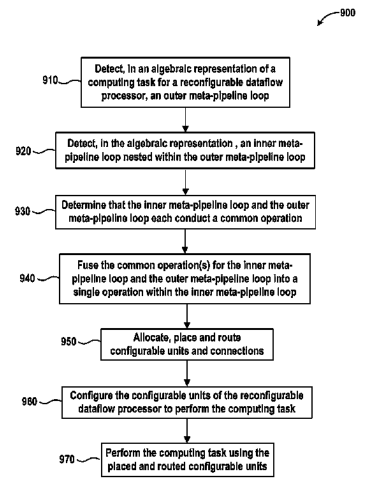
(Continued)

Primary Examiner — Satish Rampuria
(74) *Attorney, Agent, or Firm* — Flagship Patents; Sikander M. Khan

(57) **ABSTRACT**

A method for improving throughput in a reconfigurable computing system includes detecting, in an algebraic representation of a computing task for a reconfigurable dataflow processor, an outer meta-pipeline loop, detecting an inner meta-pipeline loop nested within the outer meta-pipeline loop, and determining that the inner meta-pipeline loop and the outer meta-pipeline loop each conduct a common operation. The method also includes fusing the common operation for the inner meta-pipeline loop and the outer meta-pipeline loop into a single operation within the inner meta-pipeline loop. The instances of the common operation may be fused if the output of a first instance of the common operation is the source for a second instance of the common operation. Examples of the common operation include an accumulator operation, a re-read operation, and a temporal (chip buffer synchronized) operation such as a temporal concatenation operation and a temporal slicing operation.

20 Claims, 13 Drawing Sheets


900
910
Detect, in an algebraic representation of a computing task for a reconfigurable dataflow processor, an outer meta-pipeline loop
920
Detect, in the algebraic representation, an inner meta-pipeline loop nested within the outer meta-pipeline loop
930
Determine that the inner meta-pipeline loop and the outer meta-pipeline loop each conduct a common operation
940
Fuse the common operation(s) for the inner meta-pipeline loop and the outer meta-pipeline loop into a single operation within the inner meta-pipeline loop
950
Allocate, place and route configurable units and connections
960
Configure the configurable units of the reconfigurable dataflow processor to perform the computing task
970
Perform the computing task using the placed and routed configurable units

(56) **References Cited**

U.S. PATENT DOCUMENTS

11,809,849	B1 *	11/2023	Zheng	G06N 3/048
2007/0050603	A1 *	3/2007	Vorbach	G06F 9/3802 711/138
2007/0294671	A1 *	12/2007	Demetriou	G06F 11/3664 717/124
2011/0238948	A1 *	9/2011	Vorbach	G06F 15/7867 712/15
2016/0048394	A1 *	2/2016	Vorbach	G06F 9/38585 712/32
2016/0055120	A1 *	2/2016	Vorbach	G06F 9/3001 712/17

OTHER PUBLICATIONS

Najjar, Walid A., et al. "High-level language abstraction for reconfigurable computing." Computer 36.8 (2003): pp. 63-69. (Year: 2003).*

Ayala-Rincón, Mauricio, et al. "Prototyping time-and space-efficient computations of algebraic operations over dynamically reconfigurable systems modeled by rewriting-logic." ACM Transactions on Design Automation of Electronic Systems (TODAES) 11.2 (2006): pp. 251-281. (Year: 2006).*

Hartenstein, Reiner. "A decade of reconfigurable computing: a visionary retrospective." Proceedings design, automation and test in Europe. Conference and exhibition 2001. IEEE, 2001. pp. 642-649. (Year: 2001).*

Koeplinger et al., Spatial: A Language and Compiler for Application Accelerators, PLDI '18, Jun. 18-22, 2018, Association for Computng Machinery, 16 pages.

M. Emani et al., Accelerating Scientific Applications With Sambanova Reconfigurable Dataflow Architecture, in Computing in Science & Engineering, vol. 23, No. 2, pp. 114-119, Mar. 26, 2021, [doi: 10.1109/MCSE.2021.3057203].

Podobas et al., A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective, IEEEAccess, vol. 2020. 3012084, Jul. 27, 2020, 25 pages.

Prabhakar et al., Plasticine: A Reconfigurable Architecture for Parallel Patterns, ISCA, Jun. 24-28, 2017, 14 pages.

Zhang et al., "SARA: Scaling a Reconfigurable Dataflow Accelerator," 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021, pp. 1041-1054.

* cited by examiner

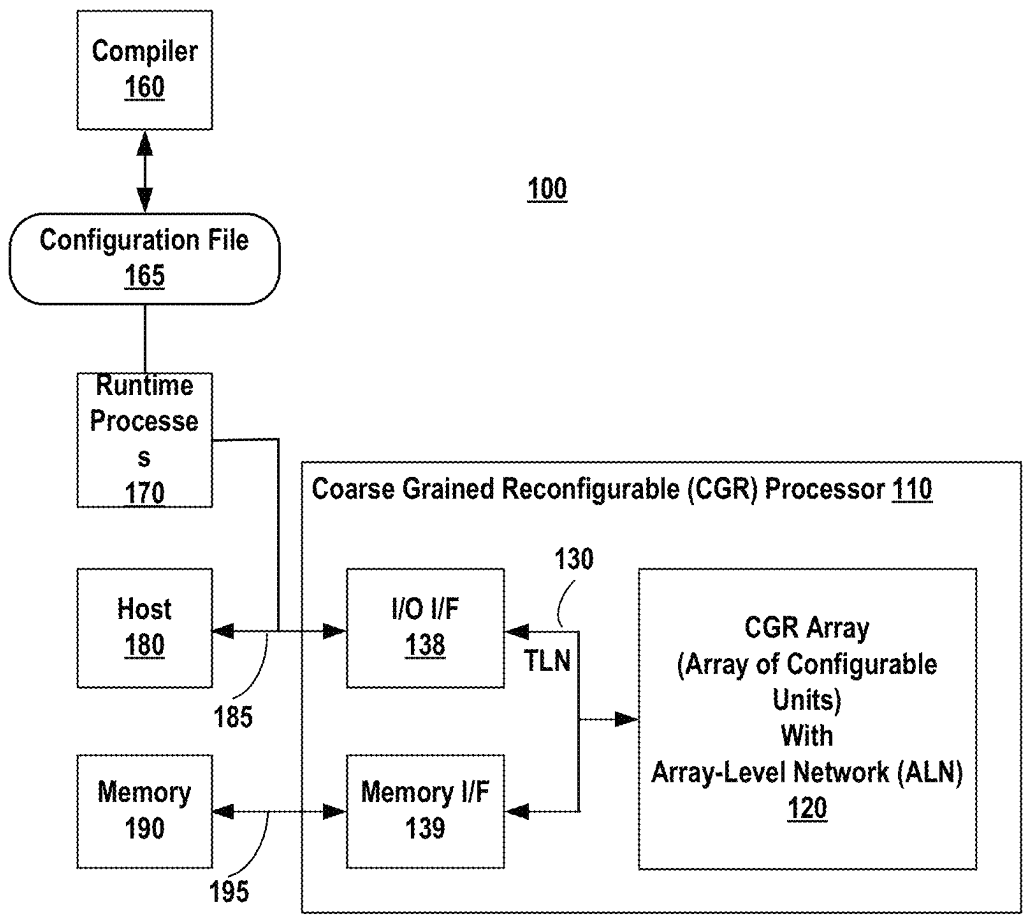

Compiler
160
100
Configuration File
165
Runtime
Processe
s
170
Coarse Grained Reconfigurable (CGR) Processor 110
130
Host
180
I/O I/F
138
TLN
CGR Array
(Array of Configurable
Units)
With
Array-Level Network (ALN)
120
185
Memory
190
Memory I/F
139
195

FIG. 1

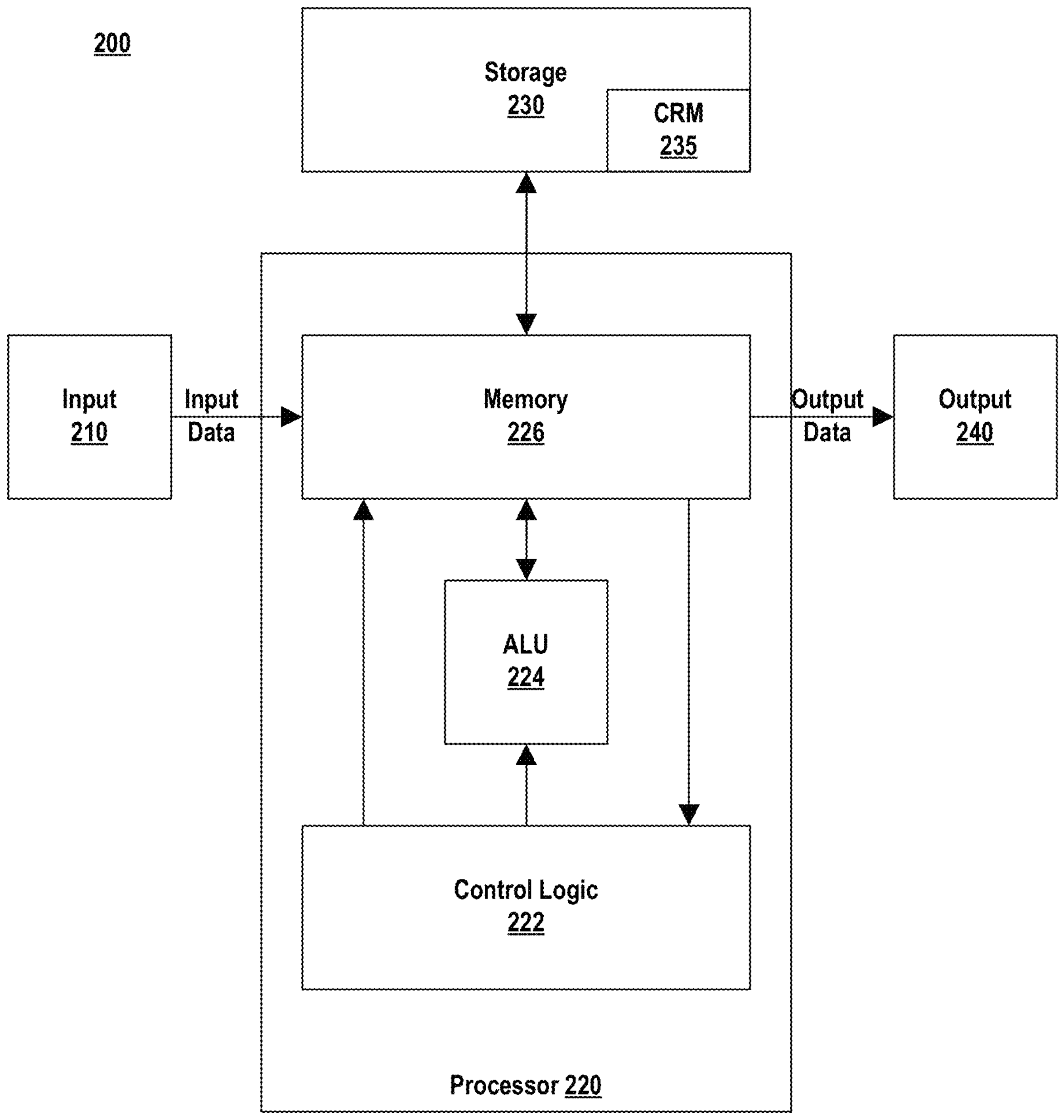

200
Storage 230
CRM 235
Input 210
Input Data
Memory 226
Output Data
Output 240
ALU 224
Control Logic 222
Processor 220

FIG. 2

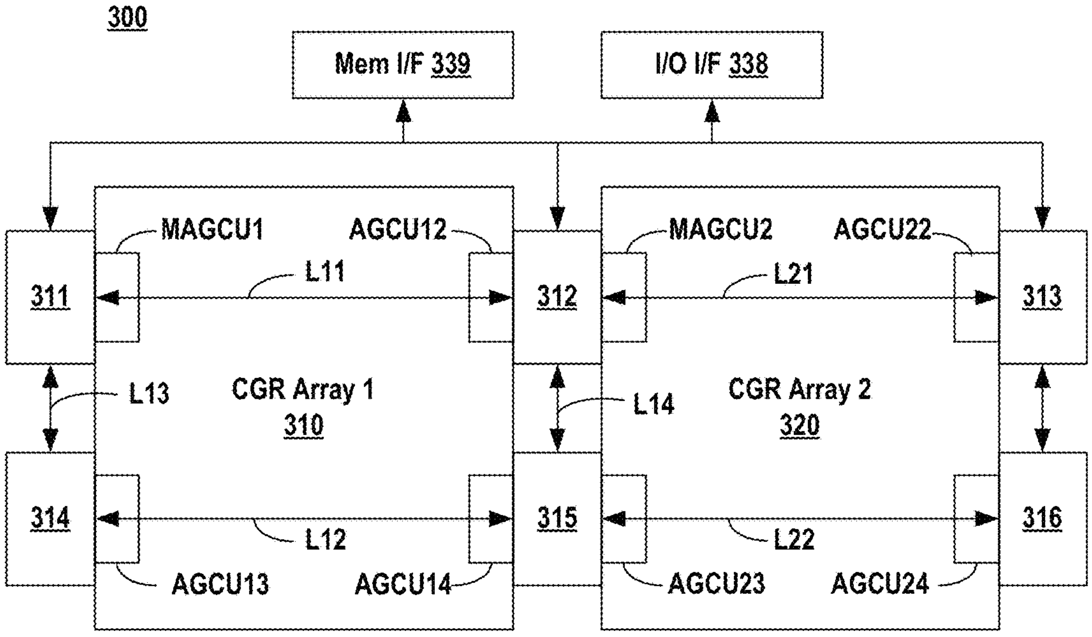

300
Mem I/F 339
I/O I/F 338
MAGCU1
AGCU12
MAGCU2
AGCU22
311
L11
312
L21
313
L13
CGR Array 1
310
L14
CGR Array 2
320
314
315
316
L12
L22
AGCU13
AGCU14
AGCU23
AGCU24

FIG. 3

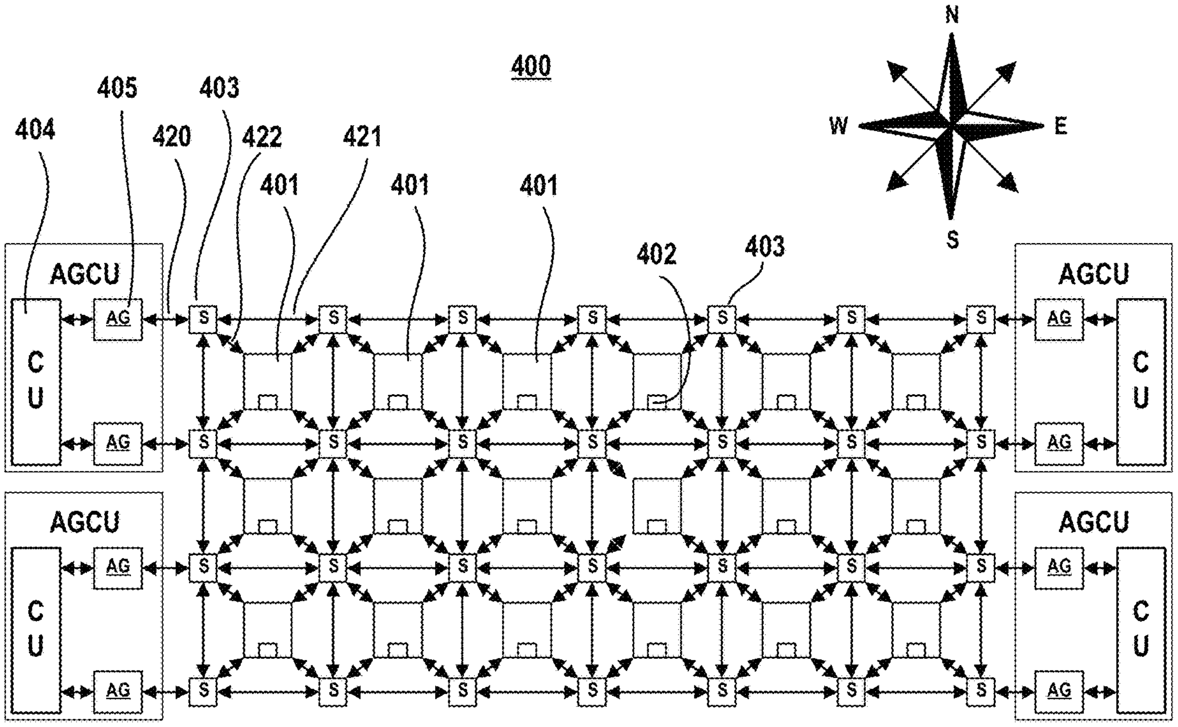

400
N
W
E
S
405
403
404
420
422
421
401
401
401
402
403
AGCU
CU
AG
S

FIG. 4

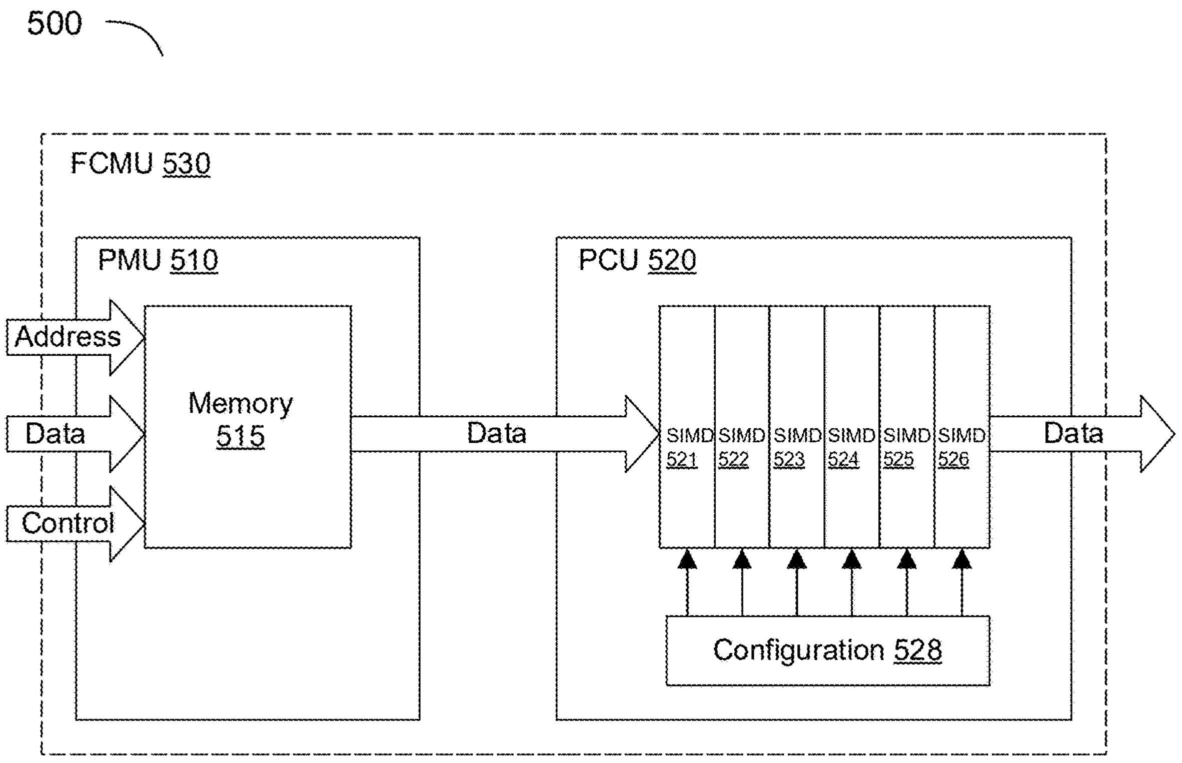

500
FCMU 530
PMU 510
PCU 520
Address
Data
Control
Memory 515
Data
SIMD 521
SIMD 522
SIMD 523
SIMD 524
SIMD 525
SIMD 526
Data
Configuration 528

Fig.5

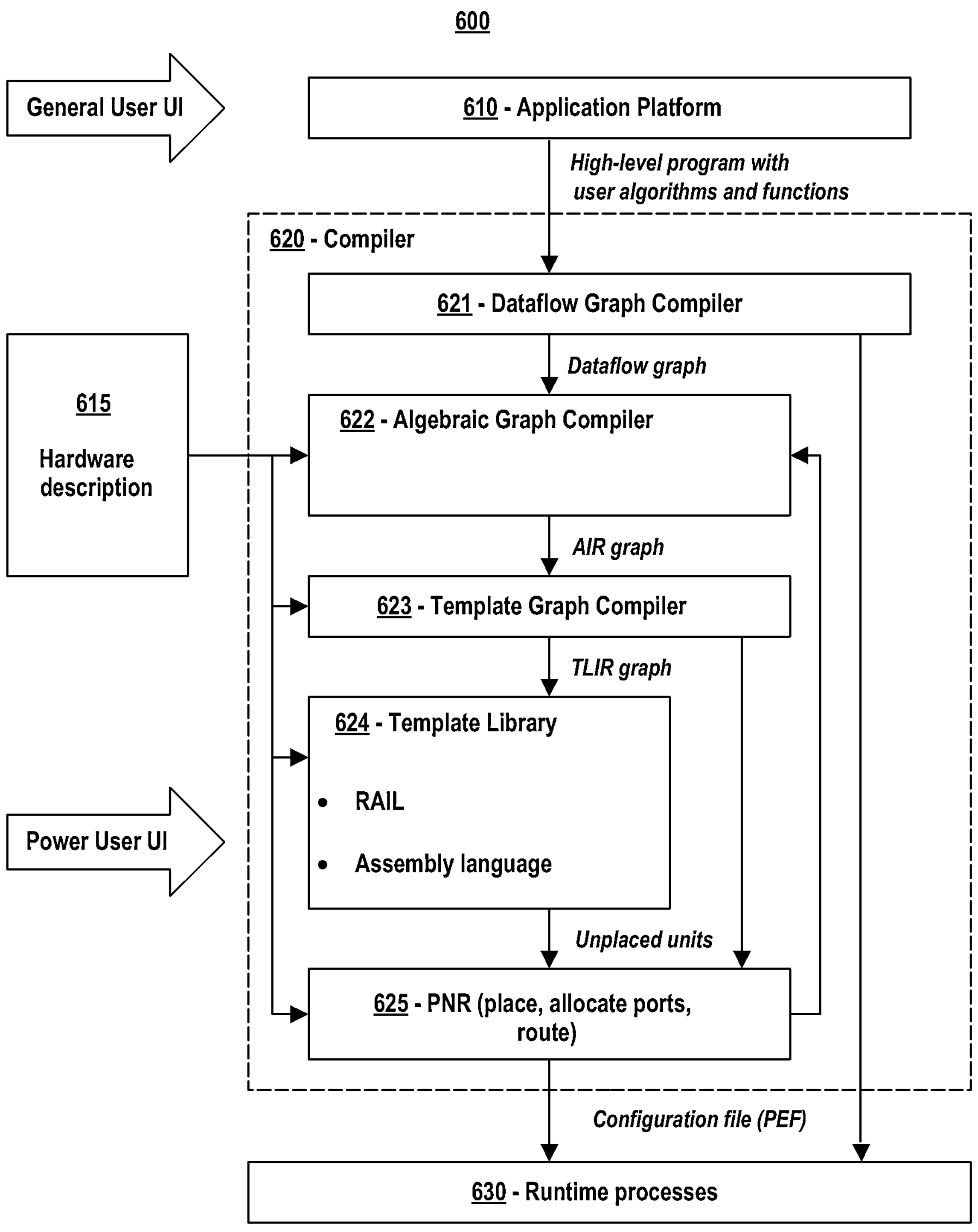

600
General User UI
610 - Application Platform
High-level program with user algorithms and functions
620 - Compiler
621 - Dataflow Graph Compiler
Dataflow graph
615
Hardware description
622 - Algebraic Graph Compiler
AIR graph
623 - Template Graph Compiler
TLIR graph
624 - Template Library
• RAIL
• Assembly language
Power User UI
Unplaced units
625 - PNR (place, allocate ports, route)
Configuration file (PEF)
630 - Runtime processes

FIG. 6

710,712

User Program Statements

```
Linear = torch.nn.linear(K,N)
Relu =    torch.nn.relu
Softmax = torch.nn.Softmax(dim=3)
X1 = torch.randn(N, C, M, K)
X2 = X1.to("RDP")
X3 = Linear(X2)
X4 = Relu(X3)
X5 = Softmax(X4)
X6 = X5.to("CPU")
```

715

User Program Compute Graph

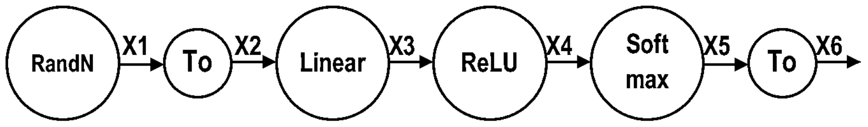


FIG. 7A

720

Air/Tensor Statements	Tensor Size	
%1 = air.Input(name: "In0"):	NxCxMxK bf16	
%2 = air.Linear(%1):	NxCxMxN bf16	
%3 = air.ReLU(%2):	NxCxMxN bf16	
%4 = air.Exp(%3):	NxCxMxN bf16	Softmax lowering
%5 = air.Sum(%4, dim: 3):	NxCxMx1 bf16	
%6 = air.Div(%4, %5):	NxCxMxN bf16	
%7 = air.Output(%6)		

725

Air/Tensor Compute Graph

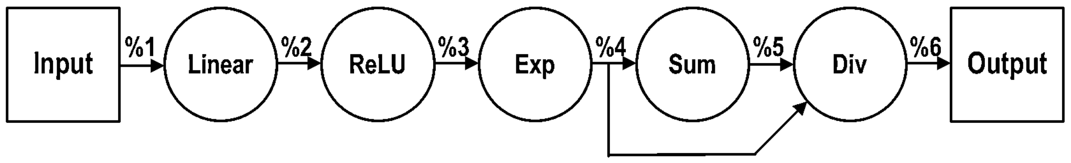


FIG. 7B

730

```
Template Dataflow Statements                    Tensor Size
%1 = tlir.Region(name: "In0"):                  NxCxMxK bf16
%2 = tlir.Region(name: "Weight"):               KxN bf16
%3 = tlir.Region(name: "Bias"):                 1xN bf16
%4 = tlir.Region(name: "Out0"):                 NxCxMxN bf16
%5 = tlir.Load(%2)
%6 = tlir.Load(%3)          732
%7 = tlir.MetaPipeline(iters: N) {
   %8 = tlir.Load(%1, dim: 1):                  CxMxK bf16
   %9 = tlir.Buffer(%5, depth: 2); 732          CxMxK bf16
   %10 = tlir.MetaPipeline(iters: C) {
      %11 = tlir.ReadSlice(%9, dim: 1):         MxK bf16
      %12 = tlir.Linear(%5, %11):               MxN bf16
      %13 = tlir.Buffer(%12, depth: 2):         MxN bf16
      %14 = tlir.AddBias(%13, %6):              MxN bf16
      %15 = air.ReLU(%14):                      MxN bf16
      %16 = air.Exp(%15):                       MxN bf16
      %17 = tlir.Buffer(%16, depth: 2):         MxN bf16
      %18 = tlir.Sum(%17, dim: 1):              Mx1 bf16
      %19 = tlir.Div(%17, %18):                 MxN bf16
      %20 = tlir.Store(%4, %19)
   }
}
```

735

Unstitched Template Compute Graph

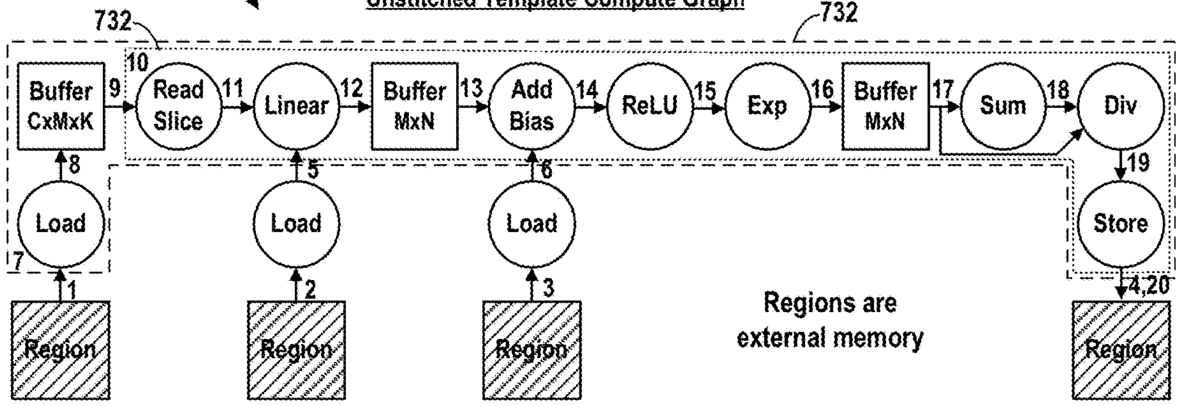


FIG. 7C

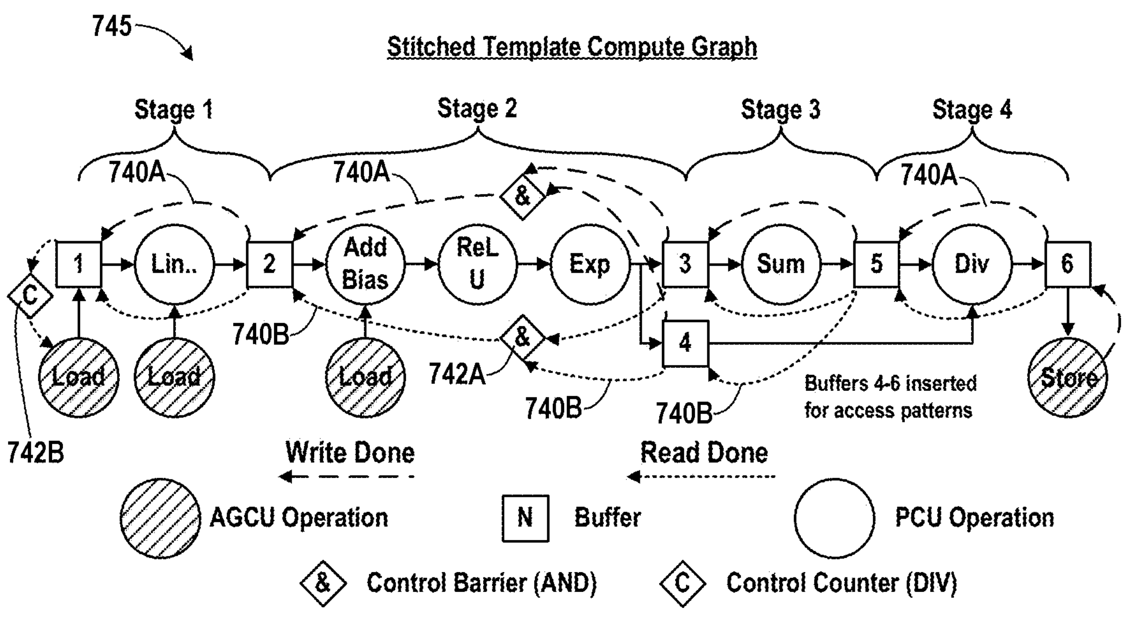

745
Stitched Template Compute Graph
Stage 1
Stage 2
Stage 3
Stage 4
740A
740A
740A
&
C
1
Lin..
2
Add Bias
ReL U
Exp
3
Sum
5
Div
6
4
&
Load
Load
Load
Store
740B
742A
740B
740B
742B
Buffers 4-6 inserted for access patterns
Write Done
Read Done
AGCU Operation
N
Buffer
PCU Operation
Control Barrier (AND)
Control Counter (DIV)

FIG. 7D

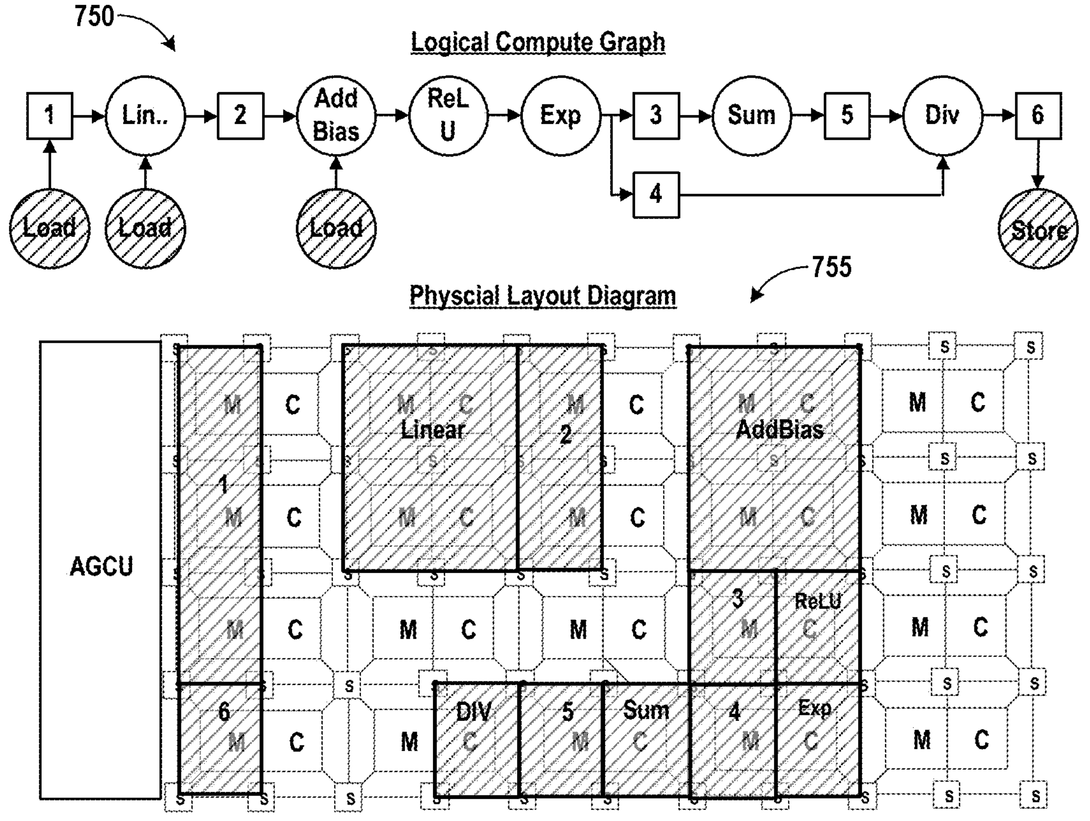

750
Logical Compute Graph
1
Lin..
2
Add Bias
ReL U
Exp
3
Sum
5
Div
6
4
Load
Load
Load
Store
755
Physcial Layout Diagram
AGCU
1
Linear
2
AddBias
3
ReLU
6
DIV
5
Sum
4
Exp
M
C
S

FIG. 7E

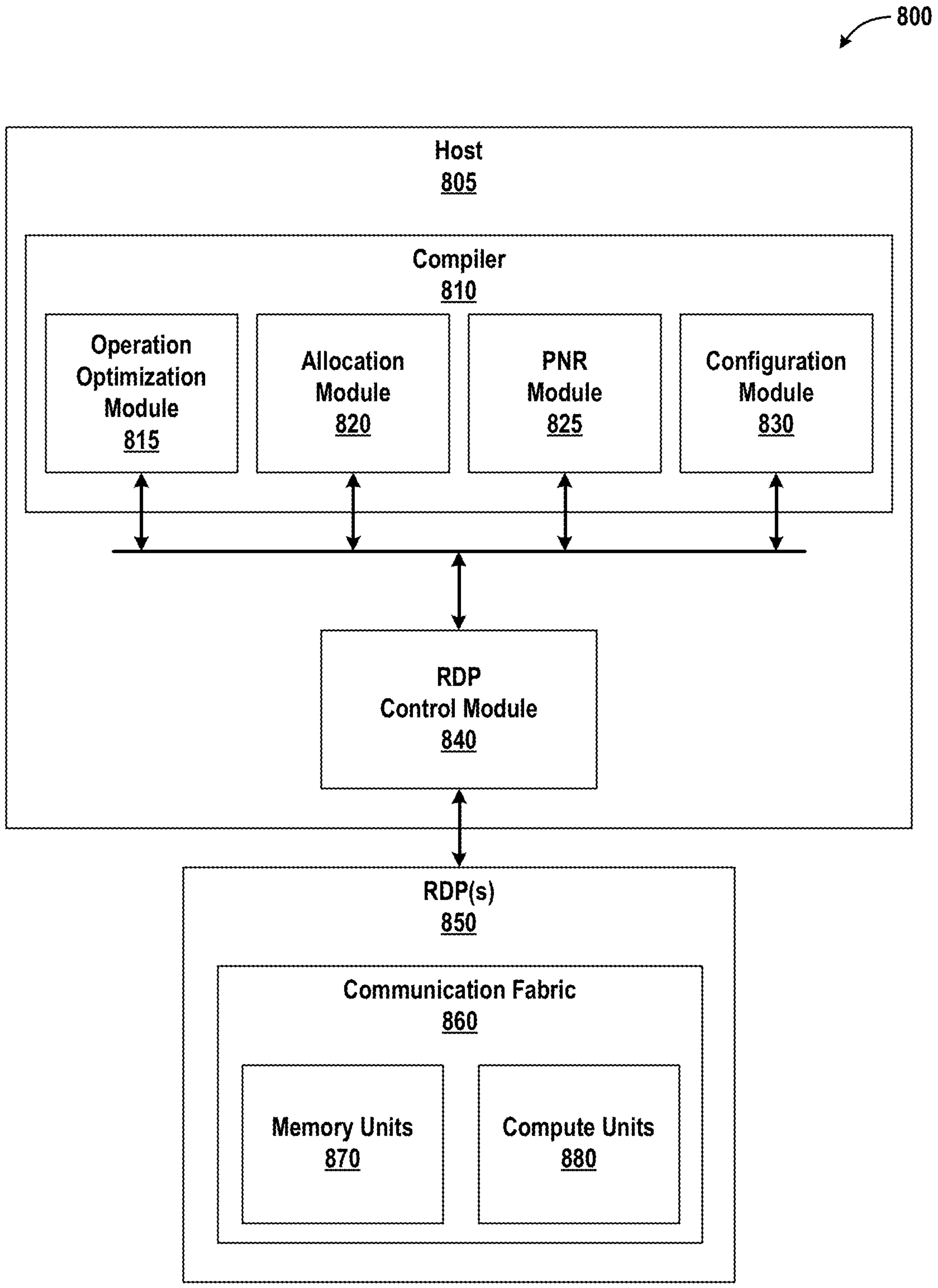

800
Host
805
Compiler
810
Operation Optimization Module
815
Allocation Module
820
PNR Module
825
Configuration Module
830
RDP Control Module
840
RDP(s)
850
Communication Fabric
860
Memory Units
870
Compute Units
880

FIG. 8

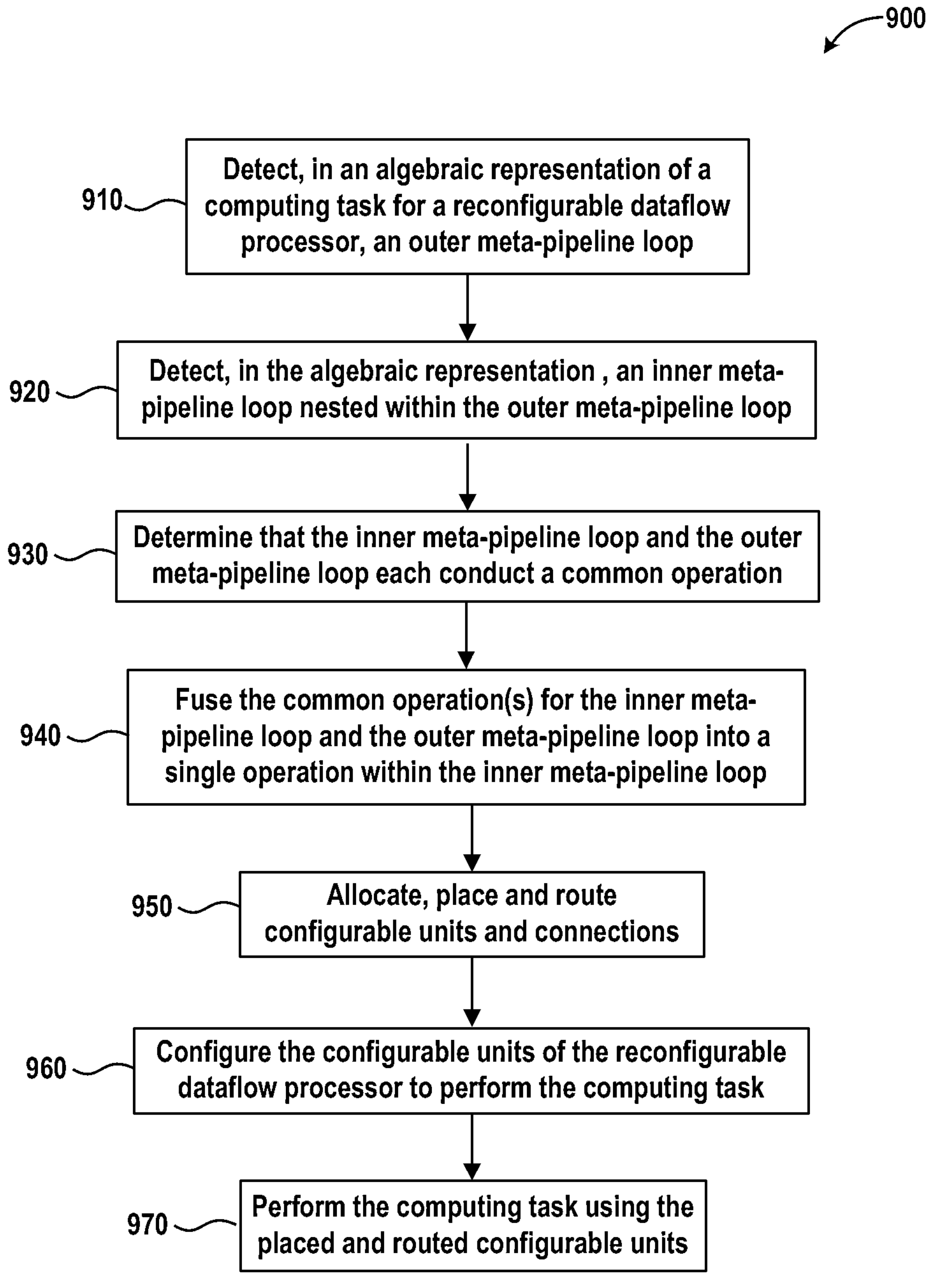
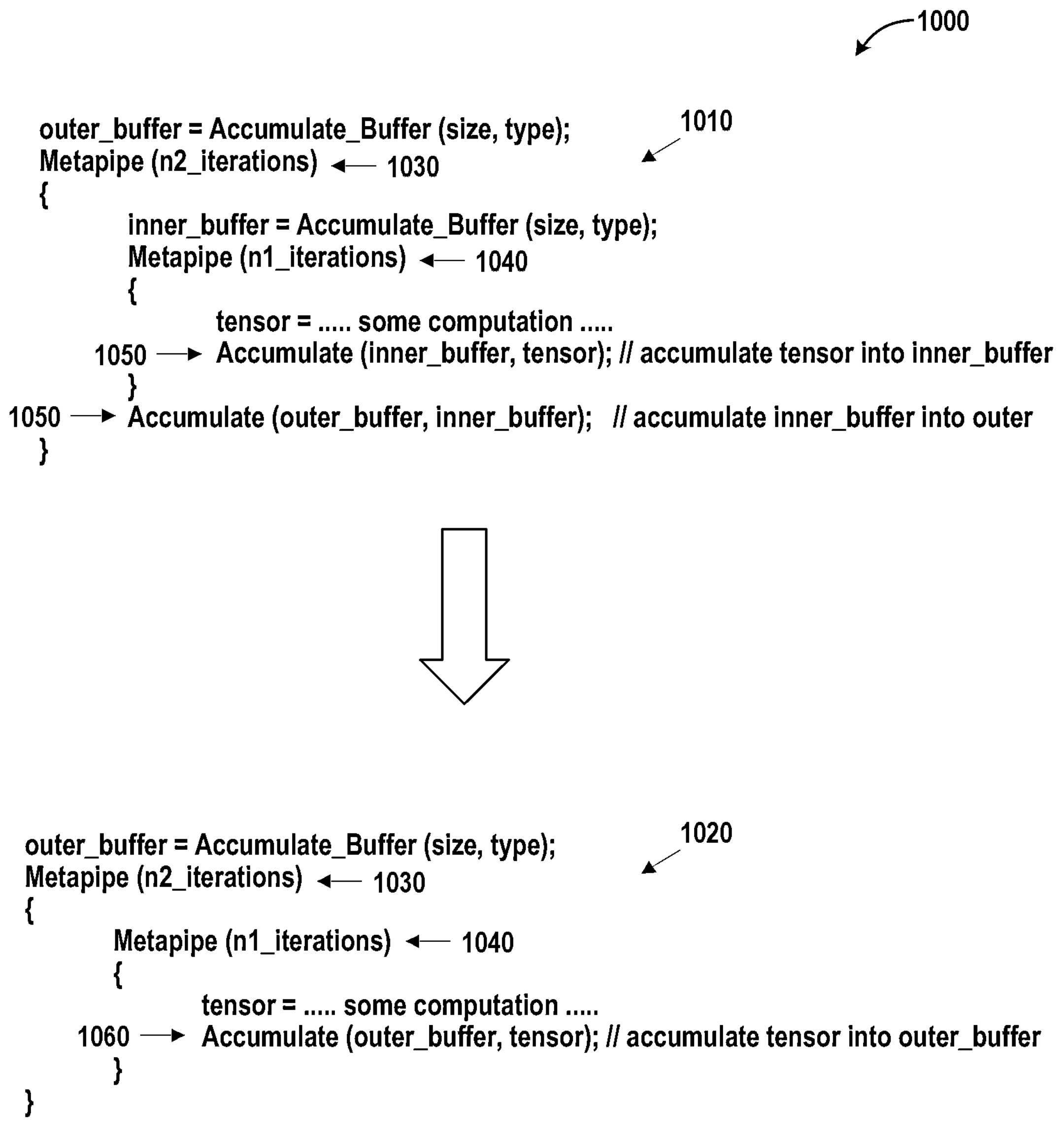

900
910
Detect, in an algebraic representation of a computing task for a reconfigurable dataflow processor, an outer meta-pipeline loop
920
Detect, in the algebraic representation , an inner meta-pipeline loop nested within the outer meta-pipeline loop
930
Determine that the inner meta-pipeline loop and the outer meta-pipeline loop each conduct a common operation
940
Fuse the common operation(s) for the inner meta-pipeline loop and the outer meta-pipeline loop into a single operation within the inner meta-pipeline loop
950
Allocate, place and route configurable units and connections
960
Configure the configurable units of the reconfigurable dataflow processor to perform the computing task
970
Perform the computing task using the placed and routed configurable units

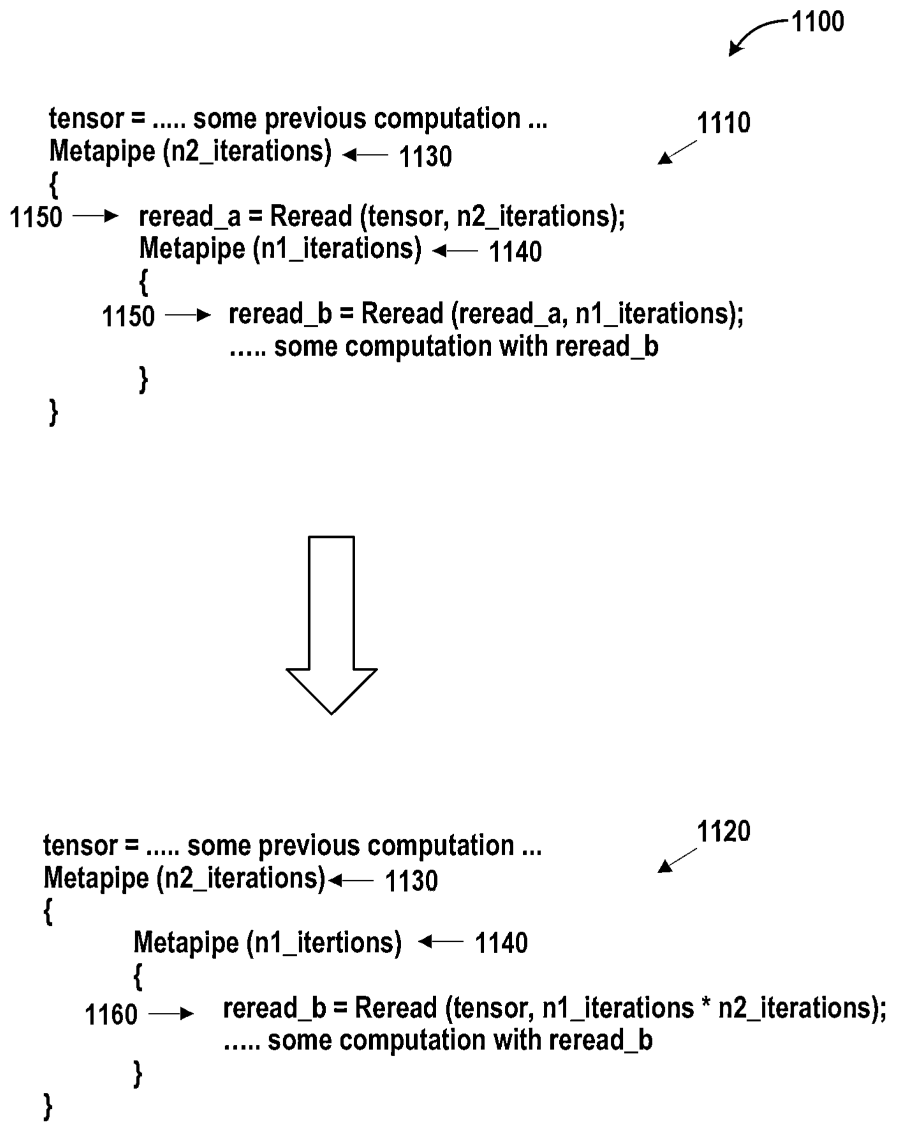
FIG. 9



```
outer_buffer = Accumulate_Buffer (size, type);
Metapipe (n2_iterations)
{
        inner_buffer = Accumulate_Buffer (size, type);
        Metapipe (n1_iterations)
        {
                tensor = ..... some computation .....
                Accumulate (inner_buffer, tensor); // accumulate tensor into inner_buffer
        }
        Accumulate (outer_buffer, inner_buffer);   // accumulate inner_buffer into outer
}
```

```
outer_buffer = Accumulate_Buffer (size, type);
Metapipe (n2_iterations)
{
        Metapipe (n1_iterations)
        {
                tensor = ..... some computation .....
                Accumulate (outer_buffer, tensor); // accumulate tensor into outer_buffer
        }
}
```

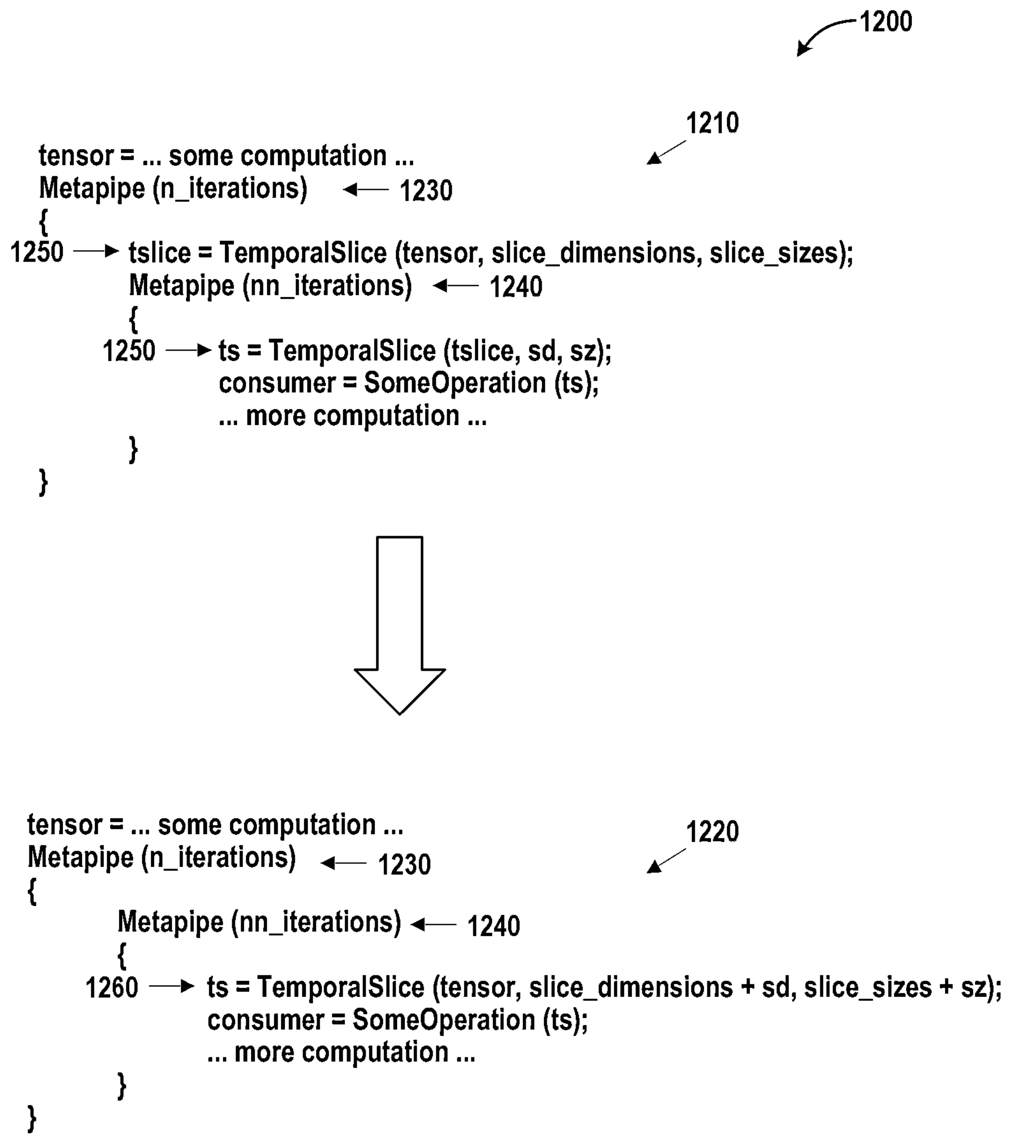
FIG. 10



```
tensor = ..... some previous computation ...
Metapipe (n2_iterations)
{
    reread_a = Reread (tensor, n2_iterations);
    Metapipe (n1_iterations)
    {
        reread_b = Reread (reread_a, n1_iterations);
        ..... some computation with reread_b
    }
}
```

```
tensor = ..... some previous computation ...
Metapipe (n2_iterations)
{
    Metapipe (n1_itertions)
    {
        reread_b = Reread (tensor, n1_iterations * n2_iterations);
        ..... some computation with reread_b
    }
}
```

FIG. 11



```
                                                              1200

                                                    1210
   tensor = ... some computation ...
   Metapipe (n_iterations)   ◄— 1230
   {
1250 —► tslice = TemporalSlice (tensor, slice_dimensions, slice_sizes);
        Metapipe (nn_iterations)  ◄— 1240
        {
    1250 —► ts = TemporalSlice (tslice, sd, sz);
            consumer = SomeOperation (ts);
            ... more computation ...
        }
   }
```

```
   tensor = ... some computation ...                1220
   Metapipe (n_iterations)  ◄— 1230
   {
        Metapipe (nn_iterations) ◄— 1240
        {
    1260 —► ts = TemporalSlice (tensor, slice_dimensions + sd, slice_sizes + sz);
            consumer = SomeOperation (ts);
            ... more computation ...
        }
   }
```

FIG. 12

OPERATION FUSION IN NESTED META-PIPELINE LOOPS

RELATED APPLICATIONS AND DOCUMENTS

This application claims the benefit of (priority to) U.S. Provisional Application 63/327,270 filed on Apr. 4, 2022 entitled "Temporal Operation Fusion."

This application is related to the following papers and commonly owned applications:

Prabhakar et al., "Plasticine: A Reconfigurable Architecture for Parallel Patterns," ISCA '17, Jun. 24-28, 2017, Toronto, ON, Canada;

Koeplinger et al., "Spatial: A Language And Compiler For Application Accelerators," Proceedings Of The 39th ACM SIGPLAN Conference On Programming Language Design And Embodiment (PLDI), Proceedings of the 43rd International Symposium on Computer Architecture, 2018;

Zhang et al., "SARA: Scaling a Reconfigurable Dataflow Accelerator," 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021, pp. 1041-1054;

U.S. Nonprovisional patent application Ser. No. 16/260,548, filed Jan. 29, 2019, entitled "MATRIX NORMAL/TRANSPOSE READ AND A RECONFIGURABLE DATA PROCESSOR INCLUDING SAME";

U.S. Nonprovisional patent application Ser. No. 15/930,381, filed May 12, 2020, entitled "COMPUTATIONALLY EFFICIENT GENERAL MATRIX-MATRIX MULTIPLICATION (GEMM)";

U.S. Nonprovisional patent application Ser. No. 16/890,841, filed Jun. 2, 2020, entitled "ANTI-CONGESTION FLOW CONTROL FOR RECONFIGURABLE PROCESSORS";

U.S. Nonprovisional patent application Ser. No. 17/023,015, filed Sep. 16, 2020, entitled "COMPILE TIME LOGIC FOR DETECTING STREAMING COMPATIBLE AND BROADCAST COMPATIBLE DATA ACCESS PATTERNS";

U.S. Nonprovisional patent application Ser. No. 17/031,679, filed Sep. 24, 2020, entitled "SYSTEMS AND METHODS FOR MEMORY LAYOUT DETERMINATION AND CONFLICT RESOLUTION";

U.S. Nonprovisional patent application Ser. No. 17/216,647, filed Mar. 29, 2021, entitled "TENSOR PARTITIONING AND PARTITION ACCESS ORDER";

U.S. Provisional Patent Application No. 63/190,749, filed May 19, 2021, entitled "FLOATING POINT MULTIPLY-ADD, ACCUMULATE UNIT WITH CARRY-SAVE ACCUMULATOR";

U.S. Provisional Patent Application No. 63/174,460, filed Apr. 13, 2021, entitled "EXCEPTION PROCESSING IN CARRY-SAVE ACCUMULATION UNIT FOR MACHINE LEARNING";

U.S. Nonprovisional patent application Ser. No. 17/397,241, filed Aug. 9, 2021, entitled "FLOATING POINT MULTIPLY-ADD, ACCUMULATE UNIT WITH CARRY-SAVE ACCUMULATOR";

U.S. Nonprovisional patent application Ser. No. 17/520,290, filed Nov. 5, 2021, entitled "SPARSE MATRIX MULTIPLIER IN HARDWARE AND A RECONFIGURABLE DATA PROCESSOR INCLUDING SAME";

All of the related application(s) and documents listed above are hereby incorporated by reference herein for all purposes.

BACKGROUND

The present subject matter relates to optimizing computing tasks for course-grained reconfigurable (CGR) processors.

Reconfigurable processors can be configured to implement a variety of functions more efficiently or faster than might be achieved using a general-purpose processor executing a computer program. For example, coarse-grain reconfigurable architectures (e.g. CGRAs) are being developed in which the configurable units in the array are more complex than used in typical, more fine-grained FPGAs, and may enable faster or more efficient (e.g., dataflow) execution of various classes of functions. For example, CGRAs have been proposed that can enable implementation of energy-efficient accelerators for machine learning and artificial intelligence workloads. See, Prabhakar, et al., "Plasticine: A Reconfigurable Architecture for Parallel Patterns," ISCA '17, Jun. 24-28, 2017, Toronto, ON, Canada.

Despite the promise of CGRAs, optimizing the compute graphs for the configurable units of a CRGA remains a challenge.

SUMMARY OF THE INVENTION

A method for improving throughput in a reconfigurable computing system includes detecting, in an algebraic representation of a computing task for a reconfigurable dataflow processor (RDP), an outer meta-pipeline loop, detecting, in the algebraic representation, an inner meta-pipeline loop nested within the outer meta-pipeline loop, and determining that the inner meta-pipeline loop and the outer meta-pipeline loop each conduct a common operation. The method also includes fusing the common operation for the inner meta-pipeline loop and the outer meta-pipeline loop into a single operation within the inner meta-pipeline loop. The instances of the common operation may be fused if the output of a first instance of the common operation is the source for a second instance of the common operation. Examples of the common operation include an accumulator operation, a re-read operation, and a temporal (chip buffer synchronized) operation such as a temporal concatenation operation and a temporal slicing operation. A corresponding system and computer program product are also disclosed herein.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** illustrates an example system including a coarse-grained reconfigurable (CGR) processor, a host, and a memory.

FIG. **2** illustrates an example of a computer, including an input device, a processor, a storage device, and an output device.

FIG. **3** illustrates example details of a CGR architecture including a top-level network (TLN) and two CGR arrays.

FIG. **4** illustrates an example CGR array, including an array of configurable nodes in an array-level network (ALN).

FIG. **5** illustrates an example of a pattern memory unit (PMU) and a pattern compute unit (PCU), which may be combined in a fused-control memory unit (FCMU).

FIG. **6** is a block diagram of a compiler stack implementation suitable for generating a configuration file for a CGR processor.

FIGS. **7**A-**7**E illustrate various representations of an example user program corresponding to various stages of a compiler stack such as the compiler stack of FIG. **6**.

FIG. **8** is a block diagram illustrating one example of a CGR dataflow computing system.

FIG. **9** is a flowchart of one example of an operation fusion method for a CGR dataflow computing system.

FIG. **10** is a code diagram of one example of operation fusion for a CGR dataflow computing system.

FIG. **11** is a code diagram of one example of operation fusion for a CGR dataflow computing system.

FIG. **12** is a code diagram of one example of operation fusion for a CGR dataflow computing system.

DETAILED DESCRIPTION

The following detailed description is made with reference to the figures. Example implementations are described to illustrate the technology disclosed, not to limit its scope, which is defined by the claims. Those of ordinary skill in the art will recognize a variety of equivalent variations on the description that follows.

FIGS. **1-7**E depict at least one example of an environment wherein the disclosed technology may be deployed while FIGS. **8-12** depict details on various examples of the disclosed technology.

Traditional compilers translate human-readable computer source code into machine code that can be executed on a Von Neumann computer architecture. In this architecture, a processor serially executes instructions in one or more threads of software code. The architecture is static, and the compiler does not determine how execution of the instructions is pipelined, or which processor or memory takes care of which thread. Thread execution is asynchronous, and safe exchange of data between parallel threads is not supported.

High-level programs for machine learning (ML) and artificial intelligence (AI) may require massively parallel computations, where many parallel and interdependent threads (meta-pipelines) exchange data. Such programs are ill-suited for execution on Von Neumann computers. They require architectures that are optimized for parallel processing, such as coarse-grained reconfigurable (CGR) architectures (CGRAs) or graphic processing units (GPUs). The ascent of ML, AI, and massively parallel architectures places new requirements on compilers, including how computation graphs, and in particular dataflow graphs, are pipelined, which operations are assigned to which compute units, how data is routed between various compute units and memory, and how synchronization is controlled particularly when a dataflow graph includes one or more nested loops, whose execution time varies dependent on the data being processed.

TERMINOLOGY

As used herein, the phrase one of should be interpreted to mean exactly one of the listed items. For example, the phrase "one of A, B, and C" should be interpreted to mean any of: only A, only B, or only C.

As used herein, the phrases at least one of and one or more of should be interpreted to mean one or more items.

Unless otherwise specified, the use of ordinal adjectives first, second, third, etc., to describe an object, merely refers to different instances or classes of the object and does not imply any ranking or sequence.

The following terms or acronyms used herein are defined at least in part as follows:

AGCU—address generator (AG) and coalescing unit (CU).

AI—artificial intelligence.

AIR—arithmetic or algebraic intermediate representation.

ALN—array-level network.

Buffer—an intermediate storage of data.

CGR—coarse-grained reconfigurable. A property of, for example, a system, a processor, an architecture (see CGRA), an array, or a unit in an array. This property distinguishes the system, etc., from field-programmable gate arrays (FPGAs), which can implement digital circuits at the gate level and are therefore fine-grained configurable.

CGRA—coarse-grained reconfigurable architecture. A data processor architecture that includes one or more arrays (CGR arrays) of CGR units.

Compiler—a translator that processes statements written in a programming language to machine language instructions for a computer processor. A compiler may include multiple stages to operate in multiple steps. Individual stages may create or update an intermediate representation (IR) of the translated statements. Compiler stages are illustrated with reference to FIG. **6**.

Computation graph—some algorithms can be represented as computation graphs. As used herein, computation graphs are a type of directed graphs comprising nodes that represent mathematical operations/expressions and edges that indicate dependencies between the operations/expressions. For example, with machine learning (ML) algorithms, input layer nodes assign variables, output layer nodes represent algorithm outcomes, and hidden layer nodes perform operations on the variables. Edges represent data (e.g., scalars, vectors, tensors) flowing between operations. In addition to dependencies, the computation graph reveals which operations and/or expressions can be executed concurrently.

CGR unit—a circuit that can be configured and reconfigured to locally store data (e.g., a memory unit or a PMU), or to execute a programmable function (e.g., a compute unit or a PCU). A CGR unit includes hardwired functionality that performs a limited number of functions used in computation graphs and dataflow graphs. Further examples of CGR units include a CU and an AG, which may be combined in an AGCU. Some implementations include CGR switches, whereas other implementations may include regular switches.

CU—coalescing unit.

Data Flow Graph—a computation graph that includes one or more loops that may be nested, and wherein nodes can send messages to nodes in earlier layers to control the dataflow between the layers.

Datapath—a collection of functional units that perform data processing operations. The functional units may include memory, multiplexers, ALUs, SIMDs, multipliers, registers, buses, etc.

FCMU—fused compute and memory unit—a circuit that includes both a memory unit and a compute unit.

Graph—a collection of nodes connected by edges. Nodes may represent various kinds of items or operations, dependent on the type of graph. Edges may represent relationships, directions, dependencies, etc.

IC—integrated circuit—a monolithically integrated circuit, i.e., a single semiconductor die which may be delivered as a bare die or as a packaged circuit. For the purposes of this document, the term integrated circuit also includes packaged circuits that include multiple semiconductor dies, stacked dies, or multiple-die substrates. Such constructions are now common in the industry, produced by the same supply chains, and for the average user often indistinguishable from monolithic circuits.

Logical CGR array or logical CGR unit—a CGR array or a CGR unit that is physically realizable, but that may not have been assigned to a physical CGR array or to a physical CGR unit on an IC.

Meta-pipeline—see pipeline.

ML—machine learning.

PCU—pattern compute unit—a compute unit that can be configured to repetitively perform a sequence of operations.

PEF—processor-executable format—a file format suitable for configuring a configurable data processor.

Pipeline—a staggered flow of operations through a chain of pipeline stages. The operations may be executed in parallel and in a time-sliced fashion. Pipelining increases overall instruction throughput. CGR processors may include pipelines at different levels. For example, a compute unit may include a pipeline at the gate level to enable correct timing of gate-level operations in a synchronous logic implementation of the compute unit, and a meta-pipeline at the graph execution level (typically a sequence of logical operations that are to be repetitively executed) that enables correct timing and loop control of node-level operations of the configured graph. Gate-level pipelines are usually hard wired and unchangeable, whereas meta-pipelines are configured at the CGR processor, CGR array level, and/or GCR unit level.

Pipeline Stages—a pipeline is divided into stages that are coupled with one another to form a pipe topology.

PMU—pattern memory unit—a memory unit that can store data according to a programmed pattern.

PNR—place and route—the assignment of logical CGR units and associated processing/operations to physical CGR units in an array, and the configuration of communication paths between the physical CGR units.

RAIL—reconfigurable dataflow processor (RDP) abstract intermediate language.

CGR Array—an array of CGR units, coupled with each other through an array-level network (ALN), and coupled with external elements via a top-level network (TLN). A CGR array can physically implement the nodes and edges of a dataflow graph.

SIMD—single-instruction multiple-data—an arithmetic logic unit (ALU) that simultaneously performs a single programmable operation on multiple data elements delivering multiple output results.

TLIR—template library intermediate representation.

TLN—top-level network.

Implementations

The architecture, configurability and dataflow capabilities of an array of CGR units enable increased compute power that supports both parallel and pipelined computation. A CGR processor, which includes one or more CGR arrays (arrays of CGR units), can be programmed to simultaneously execute multiple independent and interdependent dataflow graphs. To enable simultaneous execution, the dataflow graphs may need to be distilled from a high-level program and translated to a configuration file for the CGR processor. A high-level program is source code written in programming languages like Spatial, Python, C++, and C, and may use computation libraries for scientific computing, ML, AI, and the like. The high-level program and referenced

libraries can implement computing structures and algorithms of machine learning models like AlexNet, VGG Net, GoogleNet, ResNet, ResNeXt, RCNN, YOLO, SqueezeNet, SegNet, GAN, BERT, ELMo, USE, Transformer, and Transformer-XL.

Translation of high-level programs to executable bit files is performed by a compiler. See, for example, FIGS. **6** and **7A-7E**. While traditional compilers sequentially map operations to processor instructions, typically without regard to pipeline utilization and duration (a task usually handled by the hardware), an array of CGR units requires mapping operations to processor instructions in both space (for parallelism) and time (for synchronization of interdependent computation graphs or dataflow graphs). This requirement implies that a compiler for a CGRA must decide which operation of a computation graph or dataflow graph is assigned to which of the CGR units, and how both data and, related to the support of dataflow graphs, control information flows among CGR units, and to and from external hosts and storage. This process, known as "place and route", is one of many new challenges posed to compilers for arrays of CGR units.

FIG. **1** illustrates an example coarse-grained reconfigurable architecture (CGRA) system **100** including a coarse-grained reconfigurable (CGR) processor **110** a compiler **160**, runtime processes **170**, a host **180**, and a memory **190**. CGR processor **110** includes a CGR array such as a CGR array **120**. CGR array **120** includes an array of configurable units in an array level network. CGR processor **110** further includes an IO interface **138**, and a memory interface **139**. CGR array **120** is coupled with IO interface **138** and memory interface **139** through a data bus **130** which may be part of a top-level network (TLN). Host **180** communicates with **10** interface **138** using a system data bus **185**, and memory interface **139** communicates with memory **190** using a memory bus **195**. A configurable unit in the CGR array **120** may comprise a compute unit or a memory unit. A configurable unit in the CGR array **120** may also comprise a pattern memory unit (PMU), a pattern compute unit (PCU), or a fused-compute memory unit (FCMU). Further examples include a coalescing unit (CU) and an address generator (AG), which may be combined in an AGCU. A configurable unit may also be reconfigurable.

The configurable units in the CGR array **120** may be connected with an array-level network (ALN) to provide the circuitry for execution of a computation graph or a dataflow graph that may have been derived from a high-level program with user algorithms and functions. The high-level program may include a set of procedures, such as learning or inferencing in an artificial intelligence (AI) or machine learning (ML) system. More specifically, the high-level program may include applications, graphs, application graphs, user applications, computation graphs, control flow graphs, dataflow graphs, models, deep learning applications, deep learning neural networks, programs, program images, jobs, tasks and/or any other procedures and functions that may need serial and/or parallel processing. In some implementations, execution of the graph(s) may involve using multiple CGR processors **110**. In some implementations, CGR processor **110** may include one or more ICs. In other implementations, a single IC may span multiple CGR processors **110**. In further implementations, CGR processor **110** may include multiple arrays of configurable units **120**.

Host **180** may be, or include, a computer such as further described with reference to FIG. **2**. Host **180** runs runtime processes **170**, as further referenced herein, and may also be used to run computer programs, such as compiler **160** further described herein with reference to FIG. **9**. In some implementations, compiler **160** may run on a computer that is similar to the computer described with reference to FIG. **2** but separate from host **180**.

CGR processor **110** may accomplish computational tasks by executing a configuration file **165**. Configuration file **165** may comprise a processor-executable format file suitable for configuring a CGR array **120** of a CGR processor **110**. For the purposes of this description, a configuration file corresponds to a dataflow graph, or a translation of a dataflow graph, and may further include initialization data. Compiler **160** compiles the high-level program to provide the configuration file **165**. In some implementations described herein, a CGR array **120** is configured by programming one or more configuration stores with all or parts of the configuration file **165**. A single configuration store may be at the level of the CGR processor **110** or the CGR array **120**, or a configurable unit may include an individual configuration store. The configuration file **165** may include configuration data for the CGR array **120** and the configurable units in the CGR array **120**, and link the computation graph to the CGR array **120**. Execution of the configuration file **165** by CGR processor **110** causes the array(s) of configurable units **120** (*s*) to implement the user algorithms and functions in the dataflow graph.

CGR processor **110** can be implemented on a single integrated circuit die or on a multichip module (MCM). An IC can be packaged in a single chip module or a multichip module. An MCM is an electronic package that may comprise multiple IC dies and other devices, assembled into a single module as if it were a single device. The various dies of an MCM may be mounted on a substrate, and the bare dies of the substrate are electrically coupled to the surface or to each other using for some examples, wire bonding, tape bonding or flip-chip bonding.

FIG. **2** illustrates an example of a computer **200**, including an input device **210**, a processor **220**, a storage device **230**, and an output device **240**. Although the example computer **200** is drawn with a single processor, other implementations may have multiple processors. Input device **210** may comprise a mouse, a keyboard, a sensor, an input port (for example, a universal serial bus (USB) port), and any other input device known in the art. Output device **240** may comprise a monitor, printer, and any other output device known in the art. Furthermore, part or all of input device **210** and output device **240** may be combined in a network interface, such as a Peripheral Component Interconnect Express (PCIe) interface suitable for communicating with CGR processor **110**. Input device **210** is coupled with processor **220** to provide input data, which an implementation may store in memory **226**. Processor **220** is coupled with output device **240** to provide output data from memory **226** to output device **240**. Processor **220** further includes control logic **222**, operable to control memory **226** and arithmetic and logic unit (ALU) **224**, and to receive program and configuration data from memory **226**. Control logic **222** further controls exchange of data between memory **226** and storage device **230**. Memory **226** typically comprises memory with fast access, such as static random-access memory (SRAM), whereas storage device **230** typically comprises memory with slow access, such as dynamic random-access memory (DRAM), flash memory, magnetic disks, optical disks, and any other memory type known in the art. At least a part of the memory in storage device **230** includes a non-transitory computer-readable medium (CRM **235**), such as used for storing computer programs.

FIG. **3** illustrates example details of a CGR architecture **300** including a top-level network (TLN **330**) and two CGR arrays (CGR array **310** and CGR array **320**). A CGR array comprises an array of CGR units (e.g., PMUs, PCUs, FCMUs) coupled via an array-level network (ALN), e.g., a bus system. The ALN is coupled with the TLN **330** through several AGCUs, and consequently with I/O interface **338** (or any number of interfaces) and memory interface **339**. Other implementations may use different bus or communication architectures.

Circuits on the TLN in this example include one or more external I/O interfaces, including I/O interface **338** and memory interface **339**. The interfaces to external devices include circuits for routing data among circuits coupled with the TLN and external devices, such as high-capacity memory, host processors, other CGR processors, FPGA devices, and so on, that are coupled with the interfaces.

Each depicted CGR array has four AGCUs (e.g., MAGCU1, AGCU12, AGCU13, and AGCU14 in CGR array **310**). The AGCUs interface the TLN to the ALNs and route data from the TLN to the ALN or vice versa.

One of the AGCUs in each CGR array in this example is configured to be a master AGCU (MAGCU), which includes an array configuration load/unload controller for the CGR array. The MAGCU1 includes a configuration load/unload controller for CGR array **310**, and MAGCU2 includes a configuration load/unload controller for CGR array **320**. Some implementations may include more than one array configuration load/unload controller. In other implementations, an array configuration load/unload controller may be implemented by logic distributed among more than one AGCU. In yet other implementations, a configuration load/unload controller can be designed for loading and unloading configuration of more than one CGR array. In further implementations, more than one configuration controller can be designed for configuration of a single CGR array. Also, the configuration load/unload controller can be implemented in other portions of the system, including as a stand-alone circuit on the TLN and the ALN or ALNs.

The TLN is constructed using top-level switches (switch **311**, switch **312**, switch **313**, switch **314**, switch **315**, and switch **316**) coupled with each other as well as with other circuits on the TLN, including the AGCUs, and external I/O interface **338**. The TLN includes links (e.g., L**11**, L**12**, L**21**, L**22**) coupling the top-level switches. Data may travel in packets between the top-level switches on the links, and from the switches to the circuits on the network coupled with the switches. For example, switch **311** and switch **312** are coupled by link L**11**, switch **314** and switch **315** are coupled by link L**12**, switch **311** and switch **314** are coupled by link L**13**, and switch **312** and switch **313** are coupled by link L**21**. The links can include one or more buses and supporting control lines, including for example a chunk-wide bus (vector bus). For example, the top-level network can include data, request and response channels operable in coordination for transfer of data in any manner known in the art.

FIG. **4** illustrates an example CGR array **400**, including an array of CGR units in an ALN. CGR array **400** may include several types of CGR unit **401**, such as FCMUs, PMUs, PCUs, memory units, and/or compute units. For examples of the functions of these types of CGR units, see Prabhakar et al., "Plasticine: A Reconfigurable Architecture for Parallel Patterns", ISCA **2017**, Jun. 24-28, 2017, Toronto, ON, Canada. Each of the CGR units may include a configuration store **402** comprising a set of registers or flip-flops storing configuration data that represents the setup and/or the sequence to run a program, and that can include the number of nested loops, the limits of each loop iterator, the instructions to be executed by individual stages, the source of operands, and the network parameters for the input and output interfaces. In some implementations, each CGR unit **401** comprises an FCMU. In other implementations, the array comprises both PMUs and PCUs, or memory units and compute units, arranged in a checkerboard pattern. In yet other implementations, CGR units may be arranged in different patterns. The ALN includes switch units **403** (S), and AGCUs (each including two address generators **405** (AG) and a shared coalescing unit **404** (CU)). Switch units **403** are connected among themselves via interconnects **421** and to a CGR unit **401** with interconnects **422**. Switch units **403** may be coupled with address generators **405** via interconnects **420**. In some implementations, communication channels can be configured as end-to-end connections, and switch units **403** are CGR units. In other implementations, switches route data via the available links based on address information in packet headers, and communication channels establish as and when needed.

A configuration file may include configuration data representing an initial configuration, or starting state, of individual CGR units that execute a high-level program with user algorithms and functions. Program load is the process of setting up the configuration stores in the CGR array based on the configuration data to allow the CGR units to execute the high-level program. Program load may also require loading memory units and/or PMUs.

The ALN includes one or more kinds of physical data buses, for example a chunk-level vector bus (e.g., 512 bits of data), a word-level scalar bus (e.g., 32 bits of data), and a control bus. For instance, interconnects **421** between two switches may include a vector bus interconnect with a bus width of 512 bits, and a scalar bus interconnect with a bus width of 32 bits. A control bus can comprise a configurable interconnect that carries multiple control bits on signal routes designated by configuration bits in the CGR array's configuration file. The control bus can comprise physical lines separate from the data buses in some implementations. In other implementations, the control bus can be implemented using the same physical lines with a separate protocol or in a time-sharing procedure.

Physical data buses may differ in the granularity of data being transferred. In one implementation, a vector bus can carry a chunk that includes 16 channels of 32-bit floating-point data or 32 channels of 16-bit floating-point data (i.e., 512 bits) of data as its payload. A scalar bus can have a 32-bit payload and carry scalar operands or control information. The control bus can carry control handshakes such as tokens and other signals. The vector and scalar buses can be packet-switched, including headers that indicate a destination of individual packets and other information such as sequence numbers that can be used to reassemble a file when the packets are received out of order. Individual packet headers can contain a destination identifier that identifies the geographical coordinates of the destination switch unit (e.g., the row and column in the array), and an interface identifier that identifies the interface on the destination switch (e.g., North, South, East, West, etc.) used to reach the destination unit.

A CGR unit **401** may have four ports (as drawn) to interface with switch units **403**, or any other number of ports suitable for an ALN. Individual ports may be suitable for receiving and transmitting data, or a port may be suitable for only receiving or only transmitting data.

A switch unit, as shown in the example of FIG. **4**, may have eight interfaces. The North, South, East and West interfaces of a switch unit may be used for links between switch units using interconnects **421**. The Northeast, Southeast, Northwest and Southwest interfaces of a switch unit may each be used to make a link with an FCMU, PCU or PMU instance using one of the interconnects **422**. Two switch units in each CGR array quadrant have links to an AGCU using interconnects **420**. The AGCU coalescing unit arbitrates between the AGs and processes memory requests. Individual interfaces of a switch unit can include a vector interface, a scalar interface, and a control interface to communicate with the vector network, the scalar network, and the control network. In other implementations, a switch unit may have any number of interfaces.

During execution of a graph or subgraph in a CGR array after configuration, data can be sent via one or more switch units and one or more links between the switch units to the CGR units using the vector bus and vector interface(s) of the one or more switch units on the ALN. A CGR array may comprise at least a part of CGR array **400**, and any number of other CGR arrays coupled with CGR array **400**.

A data processing operation implemented by CGR array configuration may comprise multiple graphs or subgraphs specifying data processing operations that are distributed among and executed by corresponding CGR units (e.g., FCMUs, PMUs, PCUs, AGs, and CUs).

FIG. **5** illustrates an example **500** of a PMU **510** and a PCU **520**, which may be combined in an FCMU **530**. PMU **510** may be directly coupled to PCU **520**, or optionally via one or more switches. PMU **510** includes a scratchpad memory **515**, which may receive external data, memory addresses, and memory control information (write enable, read enable) via one or more buses included in the ALN. PCU **520** includes two or more processor stages, such as SIMD **521** through SIMD **526**, and configuration store **528**. The processor stages may include ALUs, or SIMDs, as drawn, or any other reconfigurable stages that can process data.

Individual stages in PCU **520** may also hold one or more registers (not drawn) for short-term storage of parameters. Short-term storage, for example during one to several clock cycles or unit delays, allows for synchronization of data in the PCU pipeline.

Referring now to FIG. **6** which is a block diagram of a compiler stack **600** implementation suitable for generating a configuration file for a CGR processor. Referring also to FIGS. **7**A-**7**E which illustrate various representations of an example user program **710** corresponding to various stages of a compiler stack such as the compiler stack **600**. As depicted, compiler stack **600** includes several stages to convert a high-level program (e.g., user program **710**) with statements **712** that define user algorithms and functions, e.g., algebraic expressions and functions, to configuration data for the CGR units.

Compiler stack **600** may take its input from application platform **610**, or any other source of high-level program statements suitable for parallel processing, which provides a user interface for general users. It may further receive hardware description **615**, for example defining the physical units in a reconfigurable data processor or CGRA processor. Application platform **610** may include libraries such as PyTorch, TensorFlow, ONNX, Caffe, and Keras to provide user-selected and configured algorithms. The example user program **710** depicted in FIG. **7**A comprises statements **712** that invoke various PyTorch functions.

Application platform **610** outputs a high-level program to compiler **620**, which in turn outputs a configuration file to the reconfigurable data processor or CGRA processor where it is executed in runtime processes **630**. Compiler **620** may include dataflow graph compiler **621**, which may handle a dataflow graph, algebraic graph compiler **622**, template graph compiler **623**, template library **624**, and placer and router (PNR) **625**. In some implementations, template library **624** includes RDP abstract intermediate language (RAIL) and/or assembly language interfaces for power users.

Dataflow graph compiler **621** converts the high-level program with user algorithms and functions from application platform **610** to one or more dataflow graphs. The high-level program may be suitable for parallel processing, and therefore parts of the nodes of the dataflow graphs may be intrinsically parallel unless an edge in the graph indicates a dependency. Dataflow graph compiler **621** may provide code optimization steps like false data dependency elimination, dead-code elimination, and constant folding. The dataflow graphs encode the data and control dependencies of the high-level program.

Dataflow graph compiler **621** may support programming a reconfigurable data processor at higher or lower-level programming languages, for example from an application platform **610** to C++ and assembly language. In some implementations, dataflow graph compiler **621** allows programmers to provide code that runs directly on the reconfigurable data processor. In other implementations, dataflow graph compiler **621** provides one or more libraries that include predefined functions like linear algebra operations, element-wise tensor operations, non-linearities, and reductions required for creating, executing, and profiling the dataflow graphs on the reconfigurable processors. Dataflow graph compiler **621** may provide an application programming interface (API) to enhance functionality available via the application platform **610**.

Algebraic graph compiler **622** may include a model analyzer and compiler (MAC) level that makes high-level mapping decisions for (sub-graphs of the) dataflow graph based on hardware constraints. It may support various application frontends such as Samba, JAX, and TensorFlow/HLO. Algebraic graph compiler **622** may also transform the graphs via autodiff and GradNorm, perform stitching between sub-graphs, interface with template generators for performance and latency estimation, convert dataflow graph operations to AIR operation, perform tiling, sharding (database partitioning) and other operations, and model or estimate the parallelism that can be achieved on the dataflow graphs.

Algebraic graph compiler **622** may further include an arithmetic or algebraic intermediate representation (AIR) stage that translates high-level graph and mapping decisions provided by the MAC level into explicit AIR/Tensor statements **720** and one or more corresponding algebraic graphs **725** as shown in FIG. **7**B. In the depicted example, the algebraic graph compiler replaces the Softmax function specified in the user program **710** by its constituent statements/nodes (i.e., exp, sum and div). Key responsibilities of the AIR level include legalizing the graph and mapping decisions of the MAC, expanding data parallel, tiling, metapipe, region instructions provided by the MAC, inserting stage buffers and skip buffers, eliminating redundant operations, buffers and sections, and optimizing for resource use, latency, and throughput.

Template graph compiler **623** may translate AIR statements and/or graphs into TLIR statements **730** and/or graph(s) **735** (see FIG. **7**C), optimizing for the target hardware architecture, into unplaced variable-sized units (referred to as logical CGR units) suitable for PNR **625**. Meta-pipelines **732** that enable iteration control may be allocated for sections of the TLIR statements and/or corresponding sections of the graph(s) **735**. Template graph compiler **623** may add further information (name, inputs, input names and dataflow description) for PNR **625** and make the graph physically realizable through each performed step. Template graph compiler **623** may for example provide translation of AIR graphs to specific model operation templates such as for general matrix multiplication (GeMM). An implementation may convert part or all intermediate representation operations to templates, stitch templates into the dataflow and control flow, insert necessary buffers and layout transforms, generate test data and optimize for hardware use, latency, and throughput.

Implementations may use templates for common operations. Templates may be implemented using assembly language, RAIL, or similar. RAIL is comparable to assembly language in that memory units and compute units are separately programmed, but it can provide a higher level of abstraction and compiler intelligence via a concise performance-oriented domain-specific language for CGR array templates. RAIL enables template writers and external power users to control interactions between logical compute units and memory units with high-level expressions without the need to manually program capacity splitting, register allocation, etc. The logical compute units and memory units also enable stage/register allocation, context splitting, transpose slotting, resource virtualization and mapping to multiple physical compute units and memory units (e.g., PCUs and PMUs).

Template library **624** may include an assembler that provides an architecture-independent low-level programming interface as well as optimization and code generation for the target hardware. Responsibilities of the assembler may include address expression compilation, intra-unit resource allocation and management, making a template graph physically realizable with target-specific rules, low-level architecture-specific transformations and optimizations, and architecture-specific code generation.

Referring to FIG. **7**D, the template graph compiler may also determine the control signals **740** and control gates **742** required to enable the CGR units (whether logical or physical) to coordinate dataflow between the CGR units on the communication fabric of a CGR processor. This process, sometimes referred to as stitching, produces a stitched template compute graph **745** with control signals **740** and control gates **742**. In the example depicted in FIG. **7**D, the control signals **740** include write done signals **740**A and read done signals **740**B and the control gates **742** include 'AND' gates **742**A and a counting or 'DIV' gate **742**B. The control signals **740** and control gates **742** enable coordinated dataflow between the configurable units of CGR processors such as compute units, memory units, and AGCUs.

PNR **625** translates and maps logical (i.e., unplaced physically realizable) CGR units (e.g., the nodes of the logical compute graph **750** shown in FIG. **7**E) to a physical layout (e.g., the physical layout **755** shown in FIG. **7**E) on the physical chip level e.g., a physical array of CGR units. PNR **625** also determines physical data channels to enable communication among the CGR units and between the CGR units and circuits coupled via the TLN, allocates ports on the CGR units and switches, provides configuration data and initialization data for the target hardware, and produces configuration files, e.g., processor-executable format (PEF) files. It may further provide bandwidth calculations, allocate network interfaces such as AGCUs and virtual address generators (VAGs), provide configuration data that allows

AGCUs and/or VAGs to perform address translation, and control ALN switches and data routing. PNR **625** may provide its functionality in multiple steps and may include multiple modules (not shown in FIG. **6**) to provide the multiple steps, e.g., a placer, a router, a port allocator, and a PEF file generator. PNR **625** may receive its input data in various ways. For example, it may receive parts of its input data from any of the earlier modules (dataflow graph compiler **621**, algebraic graph compiler **622**, template graph compiler **623**, and/or template library **624**). In some implementations, an earlier module, such as template graph compiler **623**, may have the task of preparing all information for PNR **625** and no other units provide PNR input data directly.

Further implementations of compiler **620** provide for an iterative process, for example by feeding information from PNR **625** back to an earlier module, so that the earlier module can execute a new compilation step in which it uses physically realized results rather than estimates of or placeholders for physically realizable circuits. For example, PNR **625** may feed information regarding the physically realized circuits back to algebraic graph compiler **622**.

Memory allocations represent the creation of logical memory spaces in on-chip and/or off-chip memories for data required to implement the dataflow graph, and these memory allocations are specified in the configuration file. Memory allocations define the type and the number of hardware circuits (functional units, storage, or connectivity components). Main memory (e.g., DRAM) may be off-chip memory, and scratchpad memory (e.g., SRAM) is on-chip memory inside a CGR array. Other memory types for which the memory allocations can be made for various access patterns and layouts include cache, read-only look-up tables (LUTs), serial memories (e.g., FIFOs), and register files.

Compiler **620** binds memory allocations to unplaced memory units and binds operations specified by operation nodes in the dataflow graph to unplaced compute units, and these bindings may be specified in the configuration data. In some implementations, compiler **620** partitions parts of a dataflow graph into memory subgraphs and compute subgraphs, and specifies these subgraphs in the PEF file. A memory subgraph may comprise address calculations leading up to a memory access. A compute subgraph may comprise all other operations in the parent graph. In one implementation, a parent graph is broken up into multiple memory subgraphs and exactly one compute subgraph. A single parent graph can produce one or more memory subgraphs, depending on how many memory accesses exist in the original loop body. In cases where the same memory addressing logic is shared across multiple memory accesses, address calculation may be duplicated to create multiple memory subgraphs from the same parent graph.

Compiler **620** generates the configuration files with configuration data (e.g., a bit stream) for the placed positions and the routed data and control networks. In one implementation, this includes assigning coordinates and communication resources of the physical CGR units by placing and routing unplaced units onto the array of CGR units while maximizing bandwidth and minimizing latency.

A first example of accelerated deep learning is using a deep learning accelerator implemented in a CGRA to train a neural network. A second example of accelerated deep learning is using the deep learning accelerator to operate a trained neural network to perform inferences. A third example of accelerated deep learning is using the deep learning accelerator to train a neural network and subsequently perform inference with any one or more of the

trained neural network, information from the trained neural network, and a variant of the same.

Examples of neural networks include fully connected neural networks (FCNNs), recurrent neural networks (RNNs), graph neural networks (GNNs), convolutional neural networks (CNNs), graph convolutional networks (GCNs), long short-term memory (LSTM) networks, autoencoders, deep belief networks, and generative adversarial networks (GANs).

An example of training a neural network is determining one or more weights associated with the neural network, such as by back-propagation in a deep learning accelerator. An example of making an inference is using a trained neural network to compute results by processing input data using the weights associated with the trained neural network. As used herein, the term 'weight' is an example of a 'parameter' as used in various forms of neural network processing. For example, some neural network learning is directed to determining parameters (e.g., through back-propagation) that are usable for performing neural network inferences.

A neural network processes data according to a dataflow graph comprising layers of neurons. Example layers of neurons include input layers, hidden layers, and output layers. Stimuli (e.g., input data) are received by an input layer of neurons and the computed results of the dataflow graph (e.g., output data) are provided by an output layer of neurons. Example hidden layers include rectified linear unit (ReLU) layers, fully connected layers, recurrent layers, graphical network layers, long short-term memory layers, convolutional layers, kernel layers, dropout layers, and pooling layers. A neural network may be conditionally and/or selectively trained. After being trained, a neural network may be conditionally and/or selectively used for inference.

Examples of ICs, or parts of ICs, that may be used as deep learning accelerators, are processors such as central processing unit (CPUs), CGR processor ICs, graphics processing units (GPUs), FPGAs, ASICs, application-specific instruction-set processor (ASIP), and digital signal processors (DSPs). The disclosed technology implements efficient distributed computing by allowing an array of accelerators (e.g., reconfigurable processors) attached to separate hosts to directly communicate with each other via buffers.

FIG. **8** is a block diagram illustrating one example of a CGR dataflow computing system. As depicted, the template configuration selection system **800** includes an operation optimization module **815**, an allocation module **820**, a place and route module **825**, a configuration module **830**, an RDP control module **840**, and one or more RDPs **850** comprising a communication fabric **860**, memory units **870** and compute units **880**. The template configuration selection system **800** enables evaluation and selection of template configurations as well as placement, routing, configuration and deployment of those configured templates on the configurable units of the reconfigurable dataflow processors (RDPs) **850**.

The depicted modules **815-840** may reside within, or be available to (e.g., within a library), a compiler **810** that executes on a host **805** and compiles computing tasks for execution on the RDPs **850**. The computing task may be represented with a compute graph and/or code statements that indicate the mathematical operations that are to be executed. The operation optimization module **815** may analyze intermediate representations of a computing task and make changes to those operations that optimize throughput and/or resource utilization while maintaining the intended results of the computing task. The allocation module **820** may allocate virtual compute units and memory units to the computing task or a portion thereof. The allocation module **815** may function in conjunction with a partitioner (not shown) that partitions the compute graph into executable sub-graphs and inserts virtual memory units (i.e., buffers) into the compute graph that enable dataflow execution of the sub-graphs on reconfigurable dataflow processors such as the RDPs **850**.

The place and route module **825** may generate multiple placement graph options corresponding to the computing task and select the placement graph that best meets the objectives and resources of the RDPs **850**. For example, in some situations throughput may be the primary objective while in other situations, minimizing consumed resources may be the primary objective. The placement graphs may specify physical compute units, memory units and switch units that correspond to the virtual units of the executable sub-graph. To reduce communication distance and latency, the specified physical compute units, memory units and switch units may be neighbors in a computing grid on an RDP **850**.

The configuration module **830** may generate configuration information for the configuration units specified in the selected placement graphs. The RDP control module **840** may communicate the configuration information to the RDPs **850** and initiate dataflow in the computing grid. The communication fabric **860** may comprise switch units (not shown) that enable communication between the RDP control module **840** and memory units **870** and compute units **880** within the RDP(s) **850**. One of skill in the art will appreciate that the placement graphs specified for execution may be relocated at runtime to a currently available RDP and/or a currently available region with a computing grid (e.g., tile) of an RDP. The relocation may preserve the relative positions and connectivity of the configurable units specified by the placement graphs and enable concurrent execution of multiple placement graphs.

FIG. **9** is a flowchart of one example of an operation fusion method **900** for a CGR dataflow computing system. As depicted, the method **900** includes detecting (**910**) an outer meta-pipeline loop, detecting (**920**) an inner meta-pipeline loop, determining (**930**) a common operation, fusing (**940**) the common operation, allocating, placing and routing (**950**) configurable units, configuring (**960**) the configurable units. The operation fusion method **900** enables increased throughput in a CGR dataflow computing system.

Detecting (**910**) the outer meta-pipeline loop may include analyzing a compute graph and/or code statements to identify a next encountered loop. Detecting (**920**) the inner meta-pipeline loop may include determining if a subsequent loop is nested within the next encountered loop. Determining (**930**) a common operation may include analyzing the operations that are within the inner meta-pipeline loop and comparing them with operations that are within the outer meta-pipeline loop and outside of the inner meta-pipeline loop. If one or more common operations are found, those operations may be fused.

Fusing (**940**) the common operation(s) may include fusing the common operation(s) for the inner meta-pipeline loop and the outer meta-pipeline loop into a single operation(s) within the inner meta-pipeline loop. Consequently, fusing (**940**) the common operation(s) may include eliminating the common operation in the outer meta-pipeline loop. Allocating, placing and routing (**950**) may include placing memory units and compute units and routing connections that enable dataflow between the memory units and compute units.

Configuring (**960**) the configurable units may include configuring the reconfigurable units of the reconfigurable computing grid. In conjunction therewith, configuring (**960**) the configurable units may include determining the configuration information for configurable units of the reconfigurable computing grid and communicating the configuration information to one or more RDPs **850** (e.g., via the RDP control module **840**). Performing (**970**) the computing task may include initiating dataflow within the reconfigurable computing grid via the RDP control module **840**.

FIG. **10** is a code diagram **1000** of one example of operation fusion for a CGR dataflow computing system. As depicted, the code diagram **1000** comprises a set of input statements **1010** that are optimized to produce a set of output statements **1020**. One of skill in the art will appreciate that the following described optimizations could be adapted for a compute graph rather than code statements.

An outer meta-pipeline loop **1030** and an inner meta-pipeline loop **1040** may be identified along with one or more common operations **1050** that are common to both loops. The depicted example shows a single pair of common operations **1050** that are fused into the fused operation **1060** within the set of output statements **1020**. In this example, the operation can be fused because the output of the first instance of the common operation (within the inner meta-pipeline loop **1040**) is the source for same operation within the second instance of the common operation (within the outer meta-pipeline loop **1030**). Fusing is accomplished by directing the output of the fused operation to the same target as the second instance of the common operation i.e., outer meta-pipeline loop **1030**.

FIG. **11** is a code diagram **1100** of one example of operation fusion for a CGR dataflow computing system. As depicted, the code diagram **1100** comprises a set of input statements **1110** that are optimized to produce a set of output statements **1120**. One of skill in the art will appreciate that the following described optimizations could be adapted for a compute graph rather than code statements.

An outer meta-pipeline loop **1130** and an inner meta-pipeline loop **1140** may be identified along with one or more common operations **1150** that are common to both loops. The depicted example shows a single pair of common operations **1150** that are fused into the fused operation **1160** within the set of output statements **1120**. In this example, the operation can be fused because the output of the first instance of the common operation (within the outer meta-pipeline loop **1130**) is the source for same operation within the second instance of the common operation (within the inner meta-pipeline loop **1140**). Fusing is accomplished by directing the output of the fused operation to the same target as the second instance of the common operation i.e., the instance within the inner meta-pipeline loop **1140**. One of skill in the art will recognize the second parameter of the fused reread operation i.e., the reread count is the product of the counts provided to the two instances of the reread operation specified in the input statements **1110**.

FIG. **12** is a code diagram of one example of operation fusion for a CGR dataflow computing system. As depicted, the code diagram **1200** comprises a set of input statements **1210** that are optimized to produce a set of output statements **1220**. One of skill in the art will appreciate that the following described optimizations could be adapted for a compute graph rather than code statements.

An outer meta-pipeline loop **1230** and an inner meta-pipeline loop **1240** may be identified along with one or more common operations **1250** that are common to both loops. The depicted example shows a single pair of common operations **1250** that are fused into the fused operation **1260** within the set of output statements **1220**. In this example, the operation can be fused because the output of the first instance of the common operation (within the outer meta-pipeline loop **1230**) is the source for same operation within the second instance of the common operation (within the inner meta-pipeline loop **1240**).

Fusing is accomplished by directing the output of the fused operation to the same target as the second instance of the common operation i.e., the instance within the inner meta-pipeline loop **1240**. One of skill in the art will recognize that the number of slice dimensions in the fused operation is the concatenation of the slice dimensions within the instances of the common operation within the input statements **1210**. Similarly, the size of the slice within the fused operation is the concatenation of the slice sizes within the instances of the common operation within the input statements **1210**.

The examples disclosed herein include a system for improving throughput in a reconfigurable computing system, the system comprising:

- a host computer comprising an optimization module configured to conduct a method comprising:
    - detecting, in an algebraic representation of a computing task for a reconfigurable dataflow processor (RDP), an outer meta-pipeline loop
    - detecting, in the algebraic representation, an inner meta-pipeline loop nested within the outer meta-pipeline loop
    - determining that the inner meta-pipeline loop and the outer meta-pipeline loop each conduct a common operation
    - fusing the common operation for the inner meta-pipeline loop and the outer meta-pipeline loop into a single operation within the inner meta-pipeline loop

Optional features for the above system include:

- wherein the output of a first instance of the common operation is the source for a second instance of the common operation
- wherein the common operation is an accumulator operation
- wherein the common operation is a re-read operation
- wherein the common operation is a temporal operation
- wherein the algebraic representation comprises a compute graph
    - wherein nodes in the compute graph correspond to tensor operations
- wherein the algebraic representation comprises code blocks
    - wherein statements in the code blocks correspond to tensor operations
- the host computer comprising an allocation module configured to allocate configurable units
    - the host computer comprising a place and route module configured to place and route the configurable units

The embodiments disclosed herein include a method for improving throughput in a reconfigurable computing system, the method comprising:

- detecting, in an algebraic representation of a computing task for a reconfigurable dataflow processor (RDP), an outer meta-pipeline loop
- detecting, in the algebraic representation, an inner meta-pipeline loop nested within the outer meta-pipeline loop
- determining that the inner meta-pipeline loop and the outer meta-pipeline loop each conduct a common operation

- fusing the common operation for the inner meta-pipeline loop and the outer meta-pipeline loop into a single operation within the inner meta-pipeline loop

Optional features for the above method include:

- wherein the output of a first instance of the common operation is the source for a second instance of the common operation
- wherein the common operation is an accumulator operation
- wherein the common operation is a re-read operation
- wherein the common operation is a temporal operation
- wherein the algebraic representation comprises a compute graph
    - wherein nodes in the compute graph correspond to tensor operations
- wherein the algebraic representation comprises code blocks
    - wherein statements in the code blocks correspond to tensor operations
- allocating, placing and routing configurable units and connections
    - wherein the configurable units comprise memory units and compute units
    - wherein the configurable units comprise switches
    - performing the computing task using the placed and routed configurable units
- configuring the RDP to perform the computing task

As will be appreciated by those of ordinary skill in the art, aspects of the various embodiments described herein may be embodied as a system, device, method, process, or computer program product apparatus. Accordingly, elements of the present disclosure may take the form of an entirely hardware embodiment, an entirely software embodiment (including firmware, resident software, micro-code, or the like) or an embodiment combining software and hardware aspects that may all generally be referred to herein as a “apparatus,” “circuit,” “circuitry,” “module,” “computer,” “logic,” “FPGA,” “unit,” “system,” or other terms. Furthermore, aspects of the various embodiments may take the form of a computer program product embodied in one or more computer-readable medium(s) having computer program code stored thereon. The phrases “computer program code” and “instructions” both explicitly include configuration information for a CGRA, an FPGA, or other programmable logic as well as traditional binary computer instructions, and the term “processor” explicitly includes logic in a CGRA, an FPGA, or other programmable logic configured by the configuration information in addition to a traditional processing core. Furthermore, “executed” instructions explicitly includes electronic circuitry of a CGRA, an FPGA, or other programmable logic performing the functions for which they are configured by configuration information loaded from a storage medium as well as serial or parallel execution of instructions by a traditional processing core.

Any combination of one or more computer-readable storage mediums may be utilized. A computer-readable storage medium may be embodied as, for example, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system, apparatus, or device, or other like storage devices known to those of ordinary skill in the art, or any suitable combination of computer-readable storage mediums described herein. In the context of this document, a computer-readable storage medium may be any tangible medium that can contain, or store, a program and/or data for use by or in connection with an instruction execution system, apparatus, or device. Even if the data in the computer-readable storage medium requires action to maintain the storage of data, such as in a traditional semiconductor-based dynamic random-access memory, the data storage in a computer-readable storage medium can be considered to be non-transitory. A computer data transmission medium, such as a transmission line, a coaxial cable, a radio-frequency carrier, and the like, may also be able to store data, although any data storage in a data transmission medium can be said to be transitory storage. Nonetheless, a computer-readable storage medium, as the term is used herein, does not include a computer data transmission medium.

Computer program code for carrying out operations for aspects of various embodiments may be written in any combination of one or more programming languages, including object-oriented programming languages such as Java, Python, C++, or the like, conventional procedural programming languages, such as the "C" programming language or similar programming languages, or low-level computer languages, such as assembly language or microcode. In addition, the computer program code may be written in VHDL, Verilog, or another hardware description language to generate configuration instructions for an FPGA, CGRA IC, or other programmable logic. The computer program code if converted into an executable form and loaded onto a computer, FPGA, CGRA IC, or other programmable apparatus, produces a computer implemented method or process. The instructions which execute on the computer, FPGA, CGRA IC, or other programmable apparatus may provide the mechanism for implementing some or all of the functions/acts specified in the flowchart and/or block diagram block or blocks. In accordance with various implementations, the computer program code may execute entirely on the user's device, partly on the user's device and partly on a remote device, or entirely on the remote device, such as a cloud-based server. In the latter scenario, the remote device may be connected to the user's device through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider). The computer program code stored in/on (i.e. embodied therewith) the non-transitory computer-readable medium produces an article of manufacture.

The computer program code, if executed by a processor, causes physical changes in the electronic devices of the processor which change the physical flow of electrons through the devices. This alters the connections between devices which changes the functionality of the circuit. For example, if two transistors in a processor are wired to perform a multiplexing operation under control of the computer program code, if a first computer instruction is executed, electrons from a first source flow through the first transistor to a destination, but if a different computer instruction is executed, electrons from the first source are blocked from reaching the destination, but electrons from a second source are allowed to flow through the second transistor to the destination. So, a processor programmed to perform a task is transformed from what the processor was before being programmed to perform that task, much like a physical plumbing system with different valves can be controlled to change the physical flow of a fluid.

What is claimed is:

1. A system for improving throughput in reconfigurable dataflow processors, the system comprising:

a host computer comprising an optimization module and configured to conduct a method of transforming a high-level program of a computing task into configuration data executable by the reconfigurable dataflow processor (RDP) including one or more arrays of configurable units, the method comprising:

detecting, in an algebraic representation of the computing task for the RDP, an outer meta-pipeline loop;

detecting, in the algebraic representation, an inner meta-pipeline loop nested within the outer meta-pipeline loop;

determining that the inner meta-pipeline loop and the outer meta-pipeline loop each conduct a common operation; and

fusing the common operation for the inner meta-pipeline loop and the outer meta-pipeline loop into a single operation within the inner meta-pipeline loop;

generating the configuration data of the computing task for the RDP processor including placement and routing of configurable units, wherein the configuration data, when loaded onto an instance of the one or more arrays of configurable units of the RDP processor, causes the one or more arrays of configurable units to implement at least the computing task; and

storing the configuration data in a non-transitory computer-readable storage medium.

2. The system of claim **1**, wherein the output of a first instance of the common operation is the source for a second instance of the common operation.

3. The system of claim **1**, wherein the common operation is one of an accumulator operation, a re-read operation and a temporal operation.

4. The system of claim **1**, wherein the algebraic representation comprises a compute graph.

5. The system of claim **4**, wherein nodes in the compute graph correspond to tensor operations.

6. The system of claim **1**, wherein the algebraic representation comprises code blocks.

7. The system of claim **6**, wherein statements in the code blocks correspond to tensor operations.

8. The system of claim **1**, wherein the host computer further comprises an allocation module configured to allocate configurable units based on the configuration data.

9. The system of claim **8**, the host computer further comprises a place and route module configured to place and route the configurable units based on the configuration data.

10. A method for improving throughput in a reconfigurable computing system, the method configured to transform a high-level program of a computing task into configuration data executable by a reconfigurable dataflow processor (RDP) including one or more arrays of configurable units and comprising:

detecting, in an algebraic representation of a computing task for the RDP, an outer meta-pipeline loop;

detecting, in the algebraic representation, an inner meta-pipeline loop nested within the outer meta-pipeline loop;

determining that the inner meta-pipeline loop and the outer meta-pipeline loop each conduct a common operation; and

fusing the common operation for the inner meta-pipeline loop and the outer meta-pipeline loop into a single operation within the inner meta-pipeline loop;

generating the configuration data of the computing task for the RDP processor including placement and routing of configurable units, wherein the configuration data, when loaded onto an instance of the one or more arrays of configurable units of the RDP processor, causes the one or more arrays of configurable units to implement at least the computing task; and storing the configuration data in a non-transitory computer-readable storage medium.

11. The method of claim **10**, wherein the output of a first instance of the common operation is the source for a second instance of the common operation.

12. The method of claim **10**, wherein the common operation is one of an accumulator operation, a re-read operation and a temporal operation.

13. The method of claim **10**, wherein the algebraic representation comprises a compute graph.

14. The method of claim **13**, wherein nodes in the compute graph correspond to tensor operations.

15. The method of claim **10**, wherein the algebraic representation comprises code blocks.

16. The method of claim **15**, wherein statements in the code blocks correspond to tensor operations.

17. The method of claim **10**, further comprising allocating, placing and routing configurable units and connections based on the configuration data.

18. The method of claim **17**, wherein the configurable units comprise one or more of memory units, compute units and switches.

19. The method of claim **10**, further comprising configuring the RDP to perform the computing task and performing the computing task.

20. A computer program product comprising a computer readable storage medium having program instructions embodied therewith, wherein the computer readable storage

medium is not a transitory signal per se, wherein the program instructions are executable by a processor to cause the processor to conduct a method of transforming a high-level program of a computing task into configuration data executable by a reconfigurable dataflow processor (RDP) including one or more arrays of configurable units, the method comprising:

detecting, in an algebraic representation of a computing task for the RDP, an outer meta-pipeline loop;

detecting, in the algebraic representation, an inner meta-pipeline loop nested within the outer meta-pipeline loop;

determining that the inner meta-pipeline loop and the outer meta-pipeline loop each conduct a common operation; and

fusing the common operation for the inner meta-pipeline loop and the outer meta-pipeline loop into a single operation within the inner meta-pipeline loop;

generating the configuration data of the computing task for the RDP processor including placement and routing of configurable units, wherein the configuration data, when loaded onto an instance of the one or more arrays of configurable units of the RDP processor, causes the one or more arrays of configurable units to implement at least the computing task; and

storing the configuration data in a non-transitory computer-readable storage medium.

* * * * *