



US 20190026250A1

(19) **United States**

(12) **Patent Application Publication**  
**Das Sarma et al.**

(10) **Pub. No.: US 2019/0026250 A1**

(43) **Pub. Date: Jan. 24, 2019**

(54) **VECTOR COMPUTATIONAL UNIT**

**Publication Classification**

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

(51) **Int. Cl.**

**G06F 15/80** (2006.01)

**G06F 9/30** (2006.01)

**G06N 99/00** (2006.01)

(72) Inventors: **Debjit Das Sarma**, San Jose, CA (US);  
**Emil Talpes**, San Mateo, CA (US);  
**Peter Joseph Bannon**, Woodside, CA (US)

(52) **U.S. Cl.**

CPC ..... **G06F 15/8053** (2013.01); **G06F 7/575** (2013.01); **G06N 99/005** (2013.01); **G06F 9/30036** (2013.01)

(21) Appl. No.: **15/920,156**

(22) Filed: **Mar. 13, 2018**

(57) **ABSTRACT**

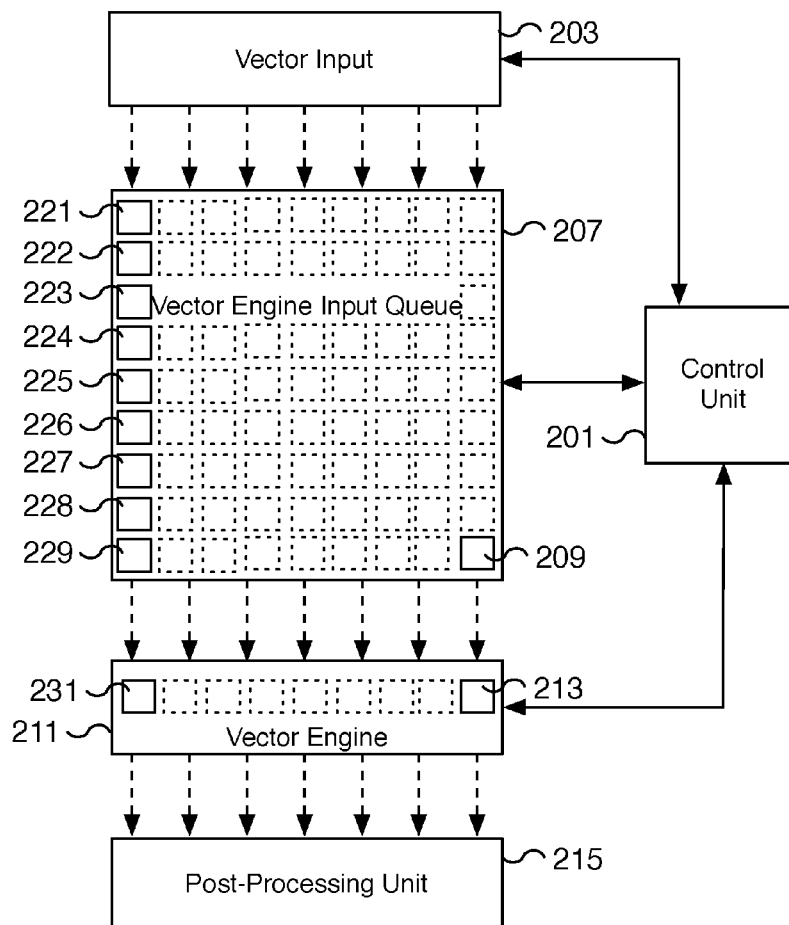
**Related U.S. Application Data**

(63) Continuation-in-part of application No. 15/710,433, filed on Sep. 20, 2017.

(60) Provisional application No. 62/625,251, filed on Feb. 1, 2018, provisional application No. 62/536,399, filed on Jul. 24, 2017, provisional application No. 62/536,399, filed on Jul. 24, 2017.

A microprocessor system comprises a computational array and a vector computational unit. The computational array includes a plurality of computation units. The vector computational unit is in communication with the computational array and includes a plurality of processing elements. The processing elements are configured to receive output data elements from the computational array and process in parallel the received output data elements.

200



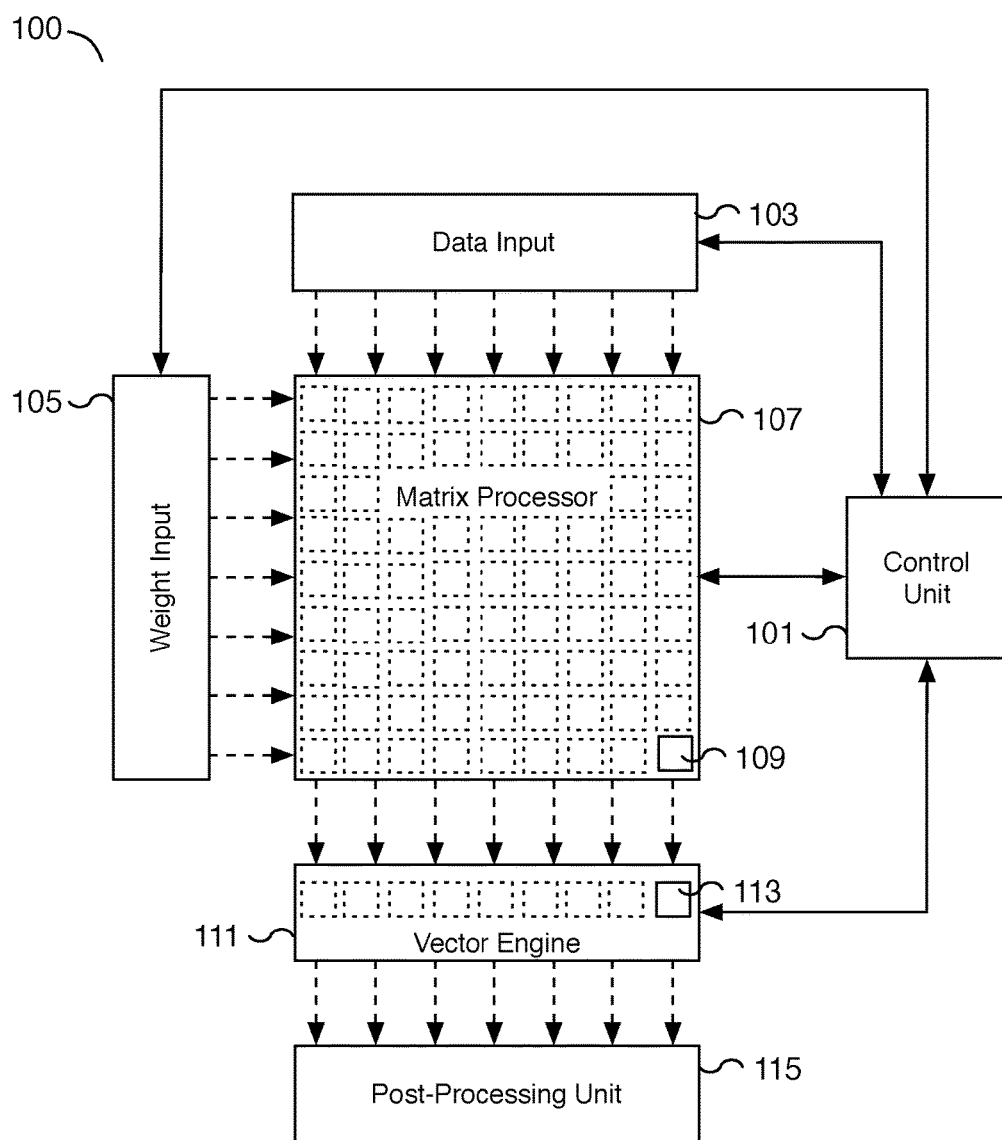


Figure 1

200

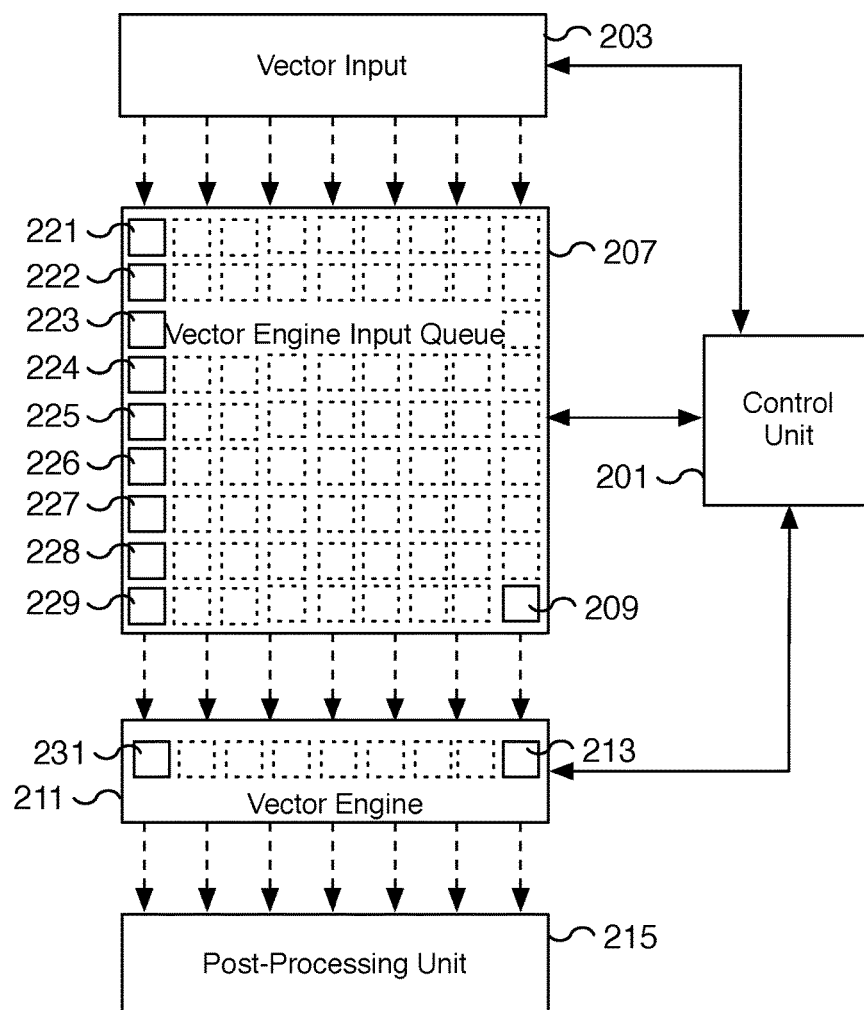


Figure 2

300

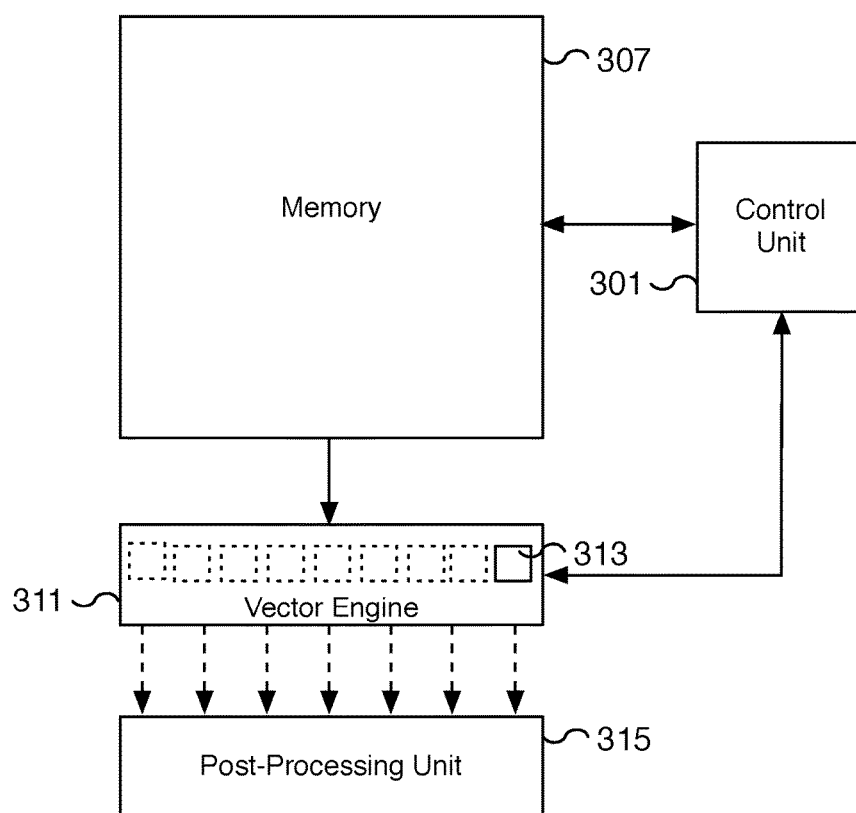


Figure 3

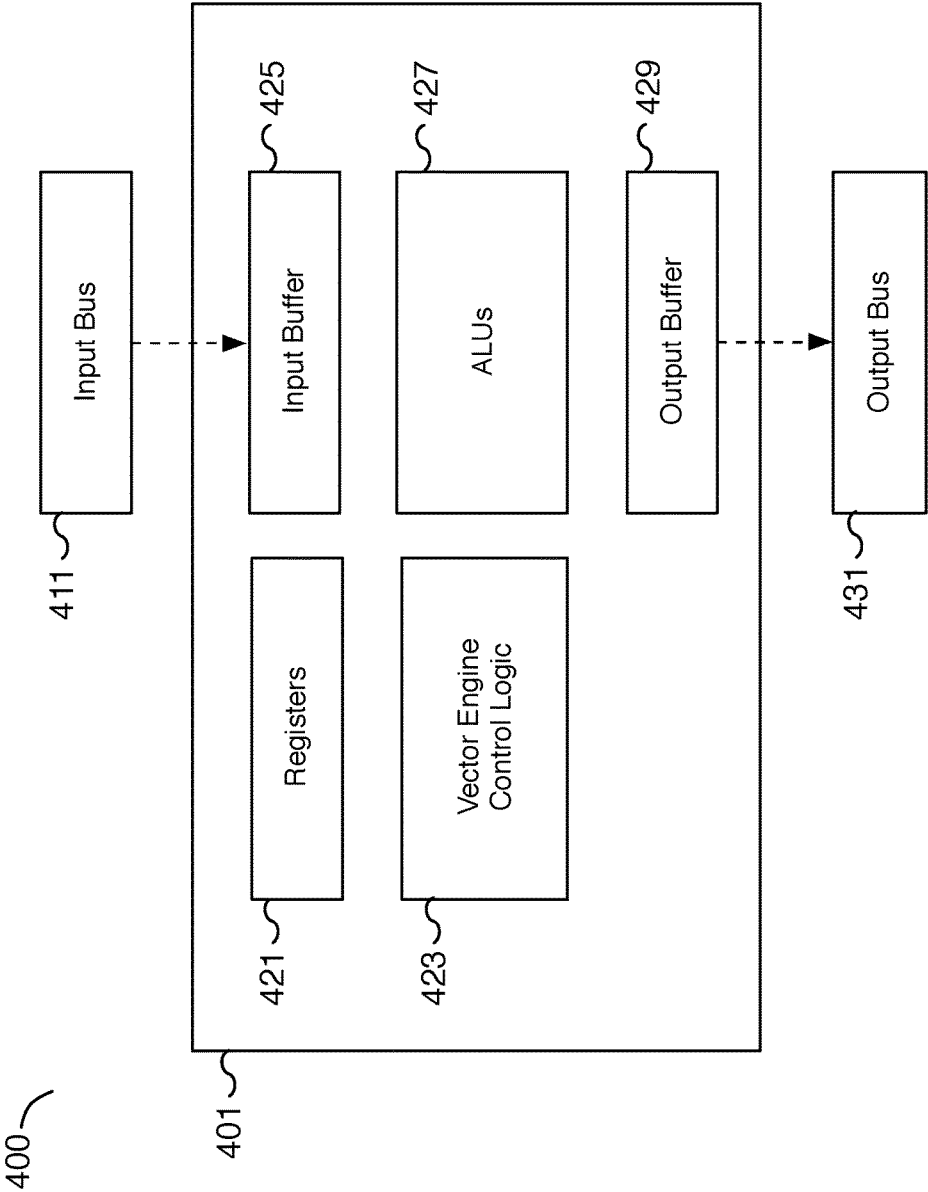


Figure 4A

Table 450

451 ~	Byte 3	Byte 2	Byte 1	Byte 0
453 ~	RD0			
455 ~	RW1		RW0	
457 ~	RB3	RB2	RB1	RB0
463 ~	RD1			
465 ~	RW3		RW2	
467 ~	RB7	RB6	RB5	RB4
473 ~	RD7			
475 ~	RW15		RW14	
477 ~	RB31	RB30	RB29	RB28

Figure 4B

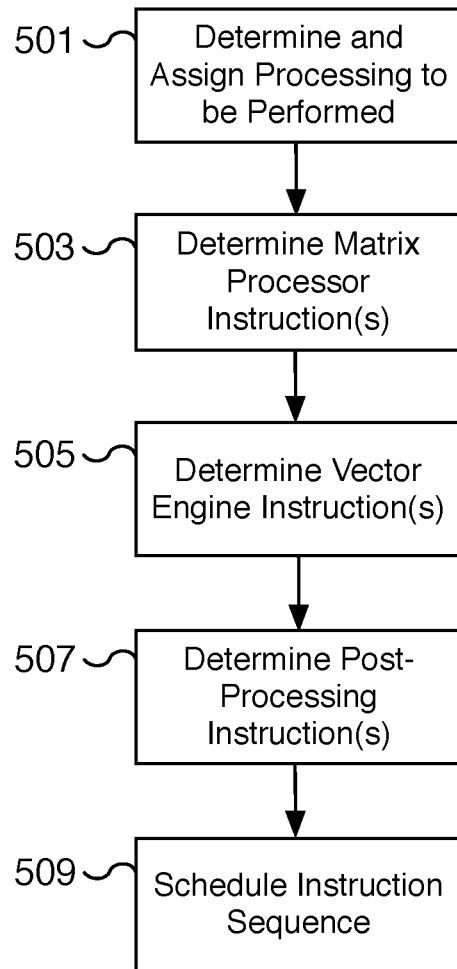


Figure 5

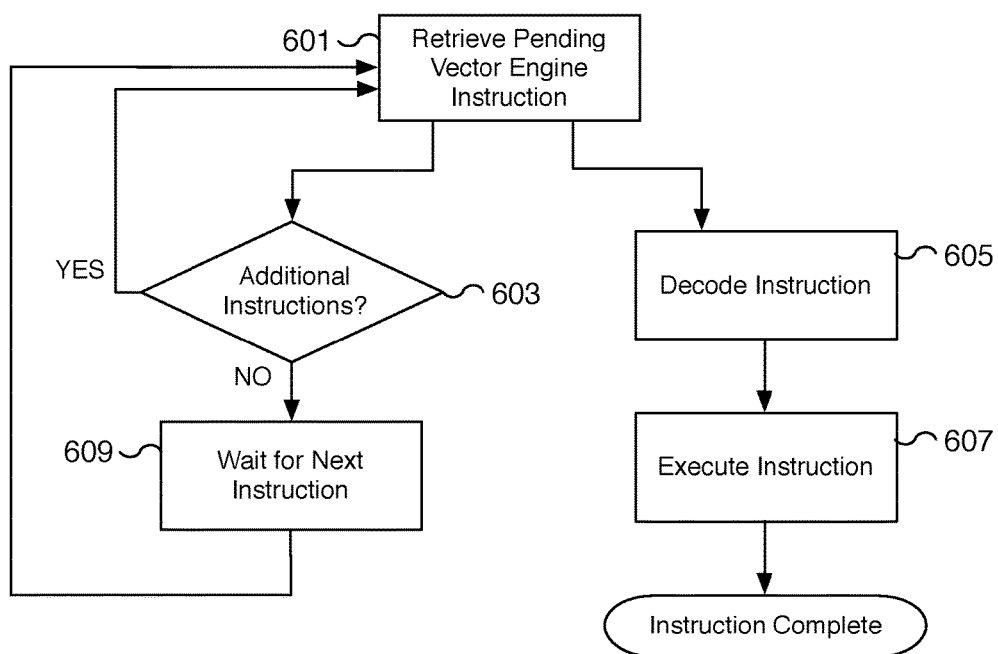


Figure 6A



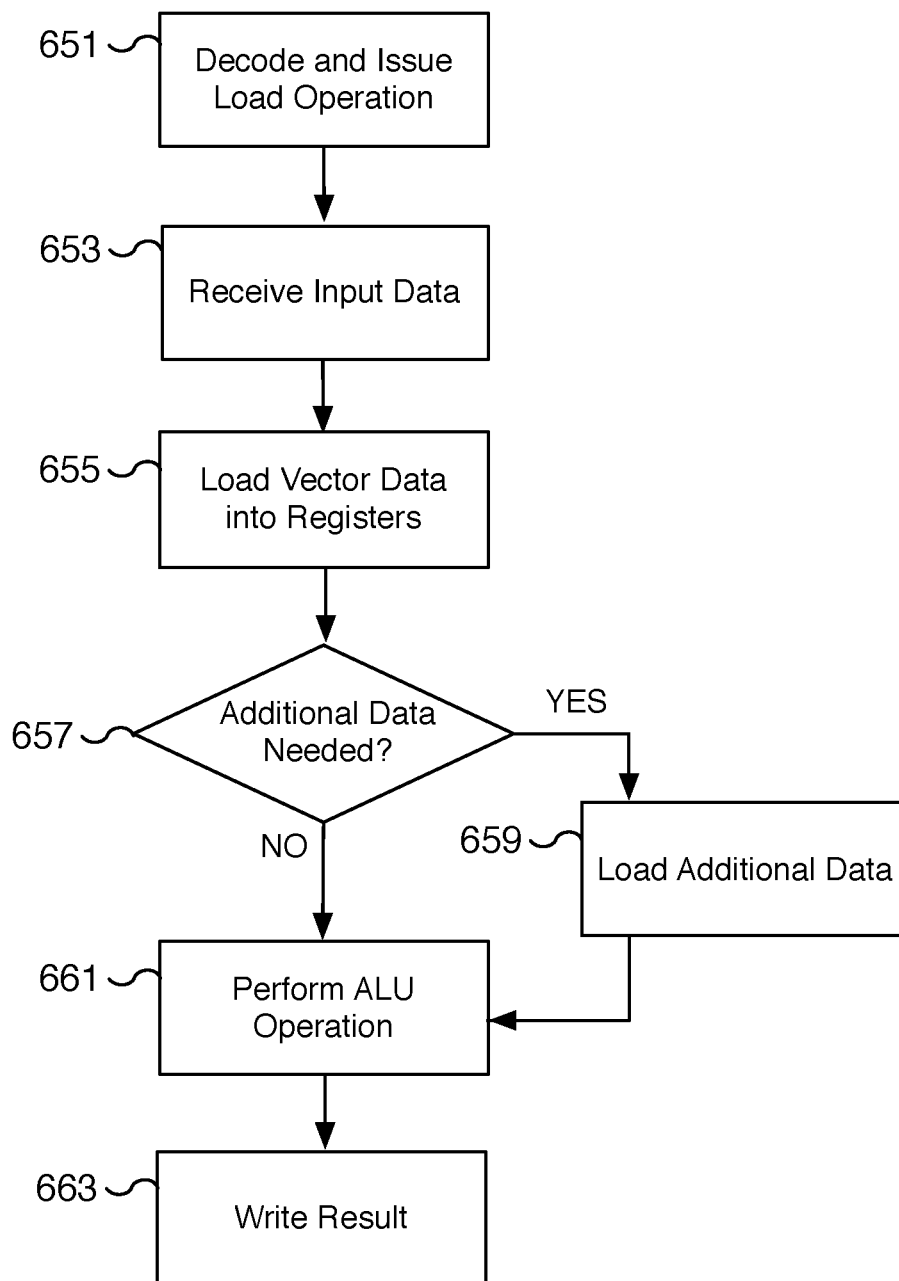


Figure 6B

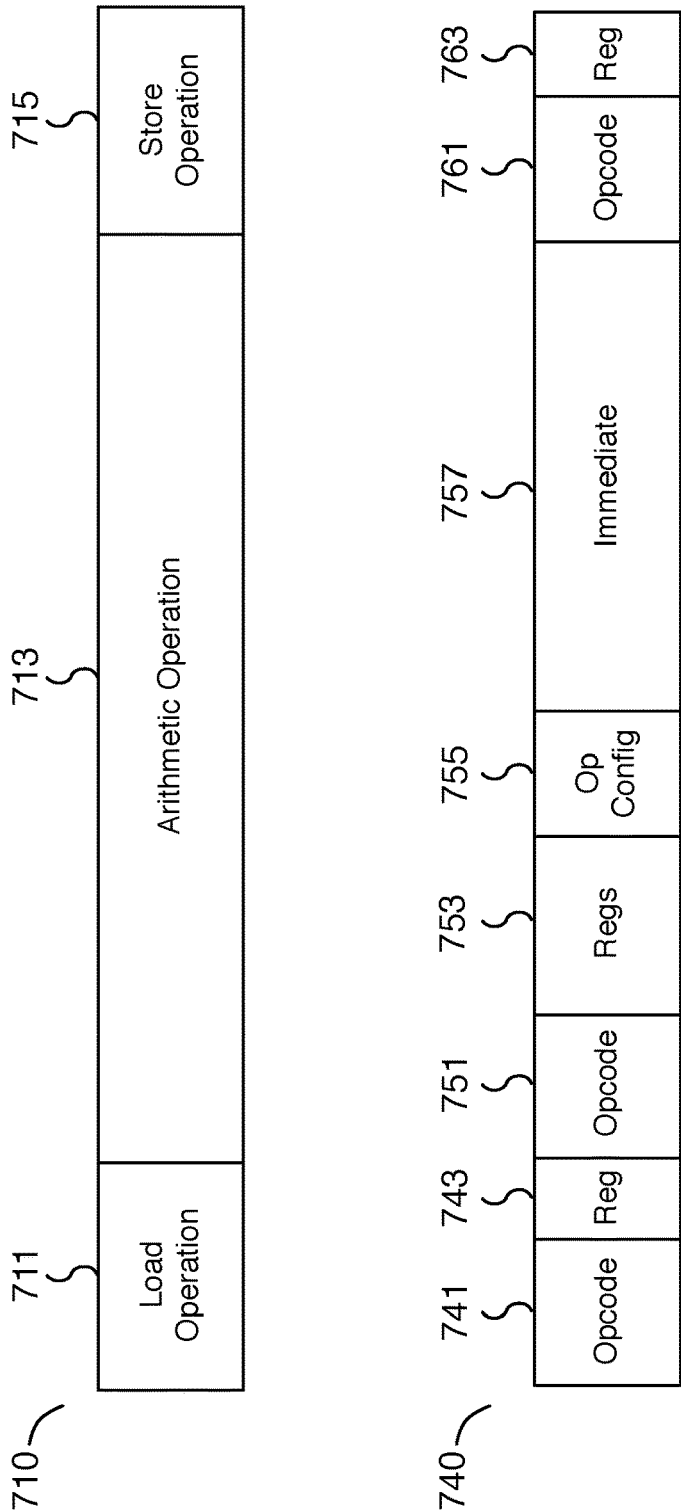


Figure 7

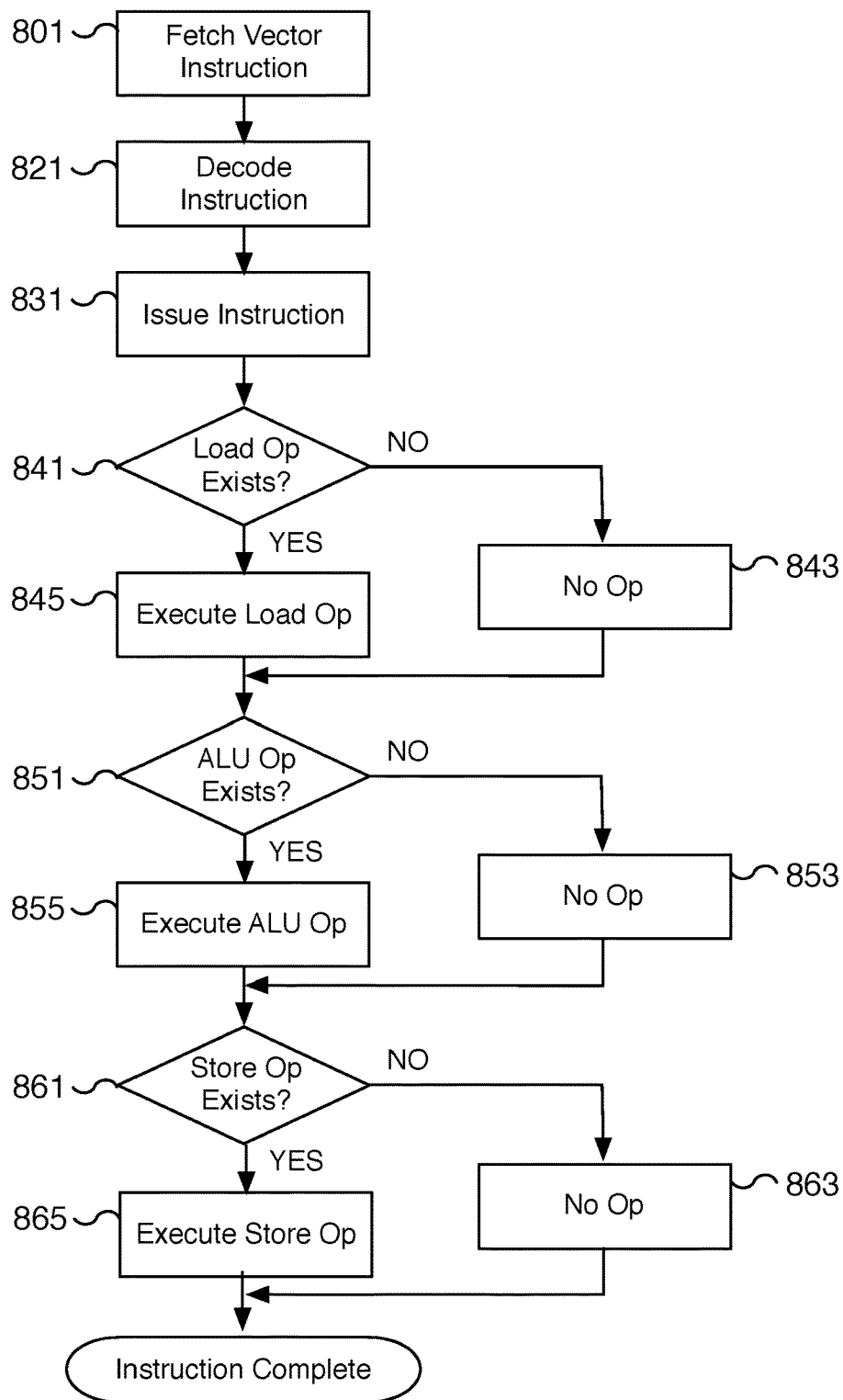


Figure 8

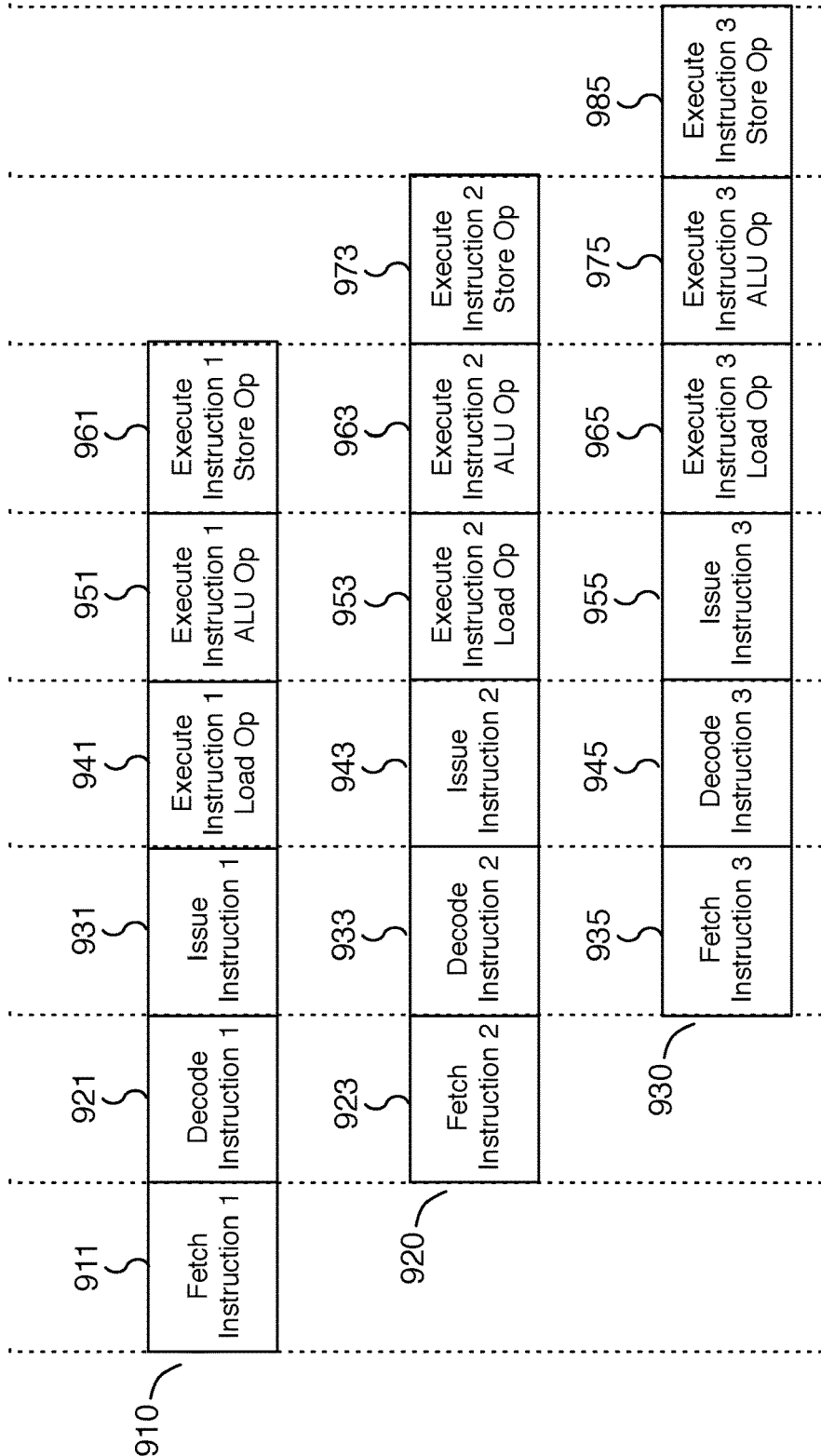


Figure 9

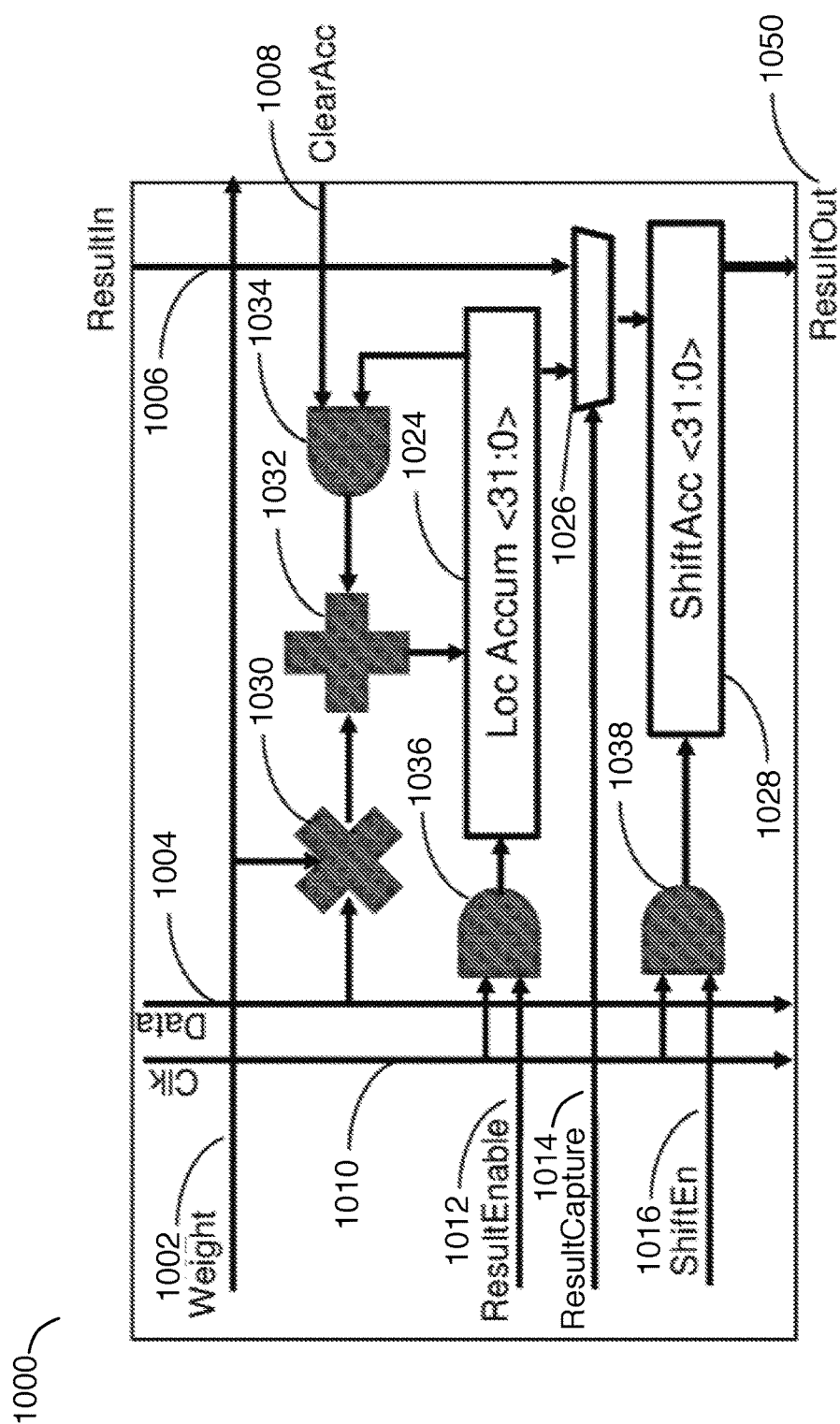


Figure 10

## VECTOR COMPUTATIONAL UNIT

### CROSS REFERENCE TO OTHER APPLICATIONS

[0001] This application claims priority to U.S. Provisional Patent Application No. 62/625,251 entitled VECTOR COMPUTATIONAL UNIT filed Feb. 1, 2018, and claims priority to U.S. Provisional Patent Application No. 62/536,399 entitled ACCELERATED MATHEMATICAL ENGINE filed Jul. 24, 2017, and is a continuation-in-part of co-pending U.S. patent application Ser. No. 15/710,433 entitled ACCELERATED MATHEMATICAL ENGINE filed Sep. 20, 2017, which claims priority to U.S. Provisional Patent Application No. 62/536,399 entitled ACCELERATED MATHEMATICAL ENGINE filed Jul. 24, 2017, all of which are incorporated herein by reference for all purposes.

### BACKGROUND OF THE INVENTION

[0002] Processing for machine learning and artificial intelligence typically requires performing mathematical operations on large sets of data and often involves solving multiple convolution layers and pooling layers. Machine learning and artificial intelligence techniques typically utilize matrix operations and non-linear functions such as activation functions. Applications of machine learning include self-driving and driver-assisted automobiles. In some scenarios, computer processors are utilized to perform machine learning training and inference. Traditional computer processors are able to perform a single mathematical operation very quickly but typically can only operate on a limited amount of data simultaneously. As an alternative, graphical processing units (GPUs) may be utilized and are capable of performing the same mathematical operations but on a larger set of data in parallel. By utilizing multiple processor cores, GPUs may perform multiple tasks in parallel and are typically capable of completing large graphics processing tasks that utilized parallelism faster than a traditional computer processor. However, neither GPUs nor traditional computer processors were originally designed for machine learning or artificial intelligence operations. Machine learning and artificial intelligence operations often rely on the repeated application of a set of specific machine learning processor operations over very large datasets. Therefore, there exists a need for a microprocessor system that supports performing machine learning and artificial intelligence specific processing operations on large datasets in parallel without the overhead of multiple processing cores for each parallel operation.

### BRIEF DESCRIPTION OF THE DRAWINGS

[0003] Various embodiments of the invention are disclosed in the following detailed description and the accompanying drawings.

[0004] FIG. 1 is a block diagram illustrating an embodiment of a microprocessor system for performing machine learning processing.

[0005] FIG. 2 is a block diagram illustrating an embodiment of a microprocessor system for performing machine learning processing.

[0006] FIG. 3 is a block diagram illustrating an embodiment of a microprocessor system for performing machine learning processing.

[0007] FIG. 4A is a block diagram illustrating an embodiment of a vector computational unit for performing machine learning processing.

[0008] FIG. 4B is a table illustrating an exemplary aliasing of vector registers.

[0009] FIG. 5 is a flow diagram illustrating an embodiment of a process for determining processor instructions for a microprocessor system.

[0010] FIG. 6A is a flow diagram illustrating an embodiment of a process for the running execution of a vector computational unit.

[0011] FIG. 6B is a flow diagram illustrating an embodiment of a process for processing vector data by a vector computational unit.

[0012] FIG. 7 is a block diagram illustrating an embodiment of an encoding format for a vector computational unit instruction.

[0013] FIG. 8 is a flow diagram illustrating an embodiment of a process for performing a single vector computational unit instruction by a vector computational unit.

[0014] FIG. 9 is a diagram illustrating an exemplary instruction cycle of a vector computational unit.

[0015] FIG. 10 is a block diagram illustrating an embodiment of a computation unit of a computational array.

### DETAILED DESCRIPTION

[0016] The invention can be implemented in numerous ways, including as a process; an apparatus; a system; a composition of matter; a computer program product embodied on a computer readable storage medium; and/or a processor, such as a processor configured to execute instructions stored on and/or provided by a memory coupled to the processor. In this specification, these implementations, or any other form that the invention may take, may be referred to as techniques. In general, the order of the steps of disclosed processes may be altered within the scope of the invention. Unless stated otherwise, a component such as a processor or a memory described as being configured to perform a task may be implemented as a general component that is temporarily configured to perform the task at a given time or a specific component that is manufactured to perform the task. As used herein, the term 'processor' refers to one or more devices, circuits, and/or processing cores configured to process data, such as computer program instructions.

[0017] A detailed description of one or more embodiments of the invention is provided below along with accompanying figures that illustrate the principles of the invention. The invention is described in connection with such embodiments, but the invention is not limited to any embodiment. The scope of the invention is limited only by the claims and the invention encompasses numerous alternatives, modifications and equivalents. Numerous specific details are set forth in the following description in order to provide a thorough understanding of the invention. These details are provided for the purpose of example and the invention may be practiced according to the claims without some or all of these specific details. For the purpose of clarity, technical material that is known in the technical fields related to the invention has not been described in detail so that the invention is not unnecessarily obscured.

[0018] A microprocessor system utilizing a vector computational unit and a vector computational unit instruction set architecture is disclosed. For example, a microprocessor

system includes a computational array in communication with a vector computational unit. In various embodiments, a computational array is a matrix processor capable of performing arithmetic operations on two input vectors and includes a plurality of computation units to receive the M operands and N operands from the input vectors. In some embodiments, the computation units are sub-circuits that include an arithmetic logic unit, an accumulator, and a shadow register for performing operations such as generating dot-products and performing various processing for convolution. Unlike conventional graphical processing unit (GPU) or central processing unit (CPU) processing cores, where each core is configured to receive its own unique processing instruction, the computation units of the computational array each perform the same computation in parallel in response to an individual instruction received by the computational array. In various embodiments, the vector computational unit includes a plurality of processing elements for performing load, arithmetic, and store operations on a vector of input data in parallel. The processing elements of the vector computational unit are configured to receive an output from the computational array. In various embodiments, the output of the computational array and the input into the vector computational unit is an array of data. The received input to the vector computational unit is processed in parallel in response to a single processor instruction. Similar to the computational array, the processing elements of the vector computational unit each perform the same computation in parallel in response to an individual instruction received by the vector computational unit. In some embodiments, the microprocessor system further includes a control unit configured to provide instructions to the vector computational unit. Each single processor instruction may specify a plurality of component instructions to be executed by the vector computational unit. In response to a single instruction, each of the plurality of processing elements of the vector computational unit processes different data elements of the vector input in parallel with the other processing elements. In some embodiments, the output of the vector computational unit is fed into a post-processing unit for performing post-processing such as pooling operations.

**[0019]** In some embodiments, a microprocessor system comprises at least a computational array and a vector computational unit. For example, a computational array is communicatively connected to a vector computational unit such that the output of the computational array is fed as input to the vector computational unit. In various embodiments, the computational array includes a plurality of computation units. For example, the computation units may be sub-circuits of a matrix processor that include the functionality for performing one or more multiply, add, and shift operations. As another example, computation units may be sub-circuits that include the functionality for performing a dot-product operation. In various embodiments, the computational array includes a sufficient number of computation units for performing multiple operations on the data inputs in parallel. For example, a computational array configured to receive M operands and N operands may include at least M×N computation units. In various embodiments, the microprocessor system further comprises a control unit for coordinating processing between the computational array and a vector computational unit. For example, the control unit may coordinate data from memory to be fed into the computational array, data from the computational array to be

fed into the vector computational unit, and/or data from the vector computational unit to be stored in memory or fed into a post-processing unit. In some embodiments, the control unit is configured to provide computational array instructions to the computational array, vector computational unit instructions to the vector computational unit, and/or post-processing instructions to a post-processing unit.

**[0020]** In some embodiments, the vector computational unit in communication with the computational array includes a plurality of processing elements configured to receive as input the output data elements from the computational array. For example, a vector computational unit, such as a vector engine, receives as input a vector for processing. The vector computational unit may include a processing element for each element of the input vector. An example vector computational unit configured to receive a vector of N elements (or operands) may include N processing elements for processing the N elements in parallel. In various embodiments, the processing elements are configured to receive output data elements from the computational array. For example, the output from the computational array may be a vector of data elements that are fed to be received by the processing elements of the vector computational unit. In various embodiments, each vector computational unit processes in parallel the received output data elements from the computational array in response to a single processor instruction. For example, a single processor instruction is applied to each of the processing elements of the vector computational unit to be performed on the corresponding data element.

**[0021]** In some embodiments, a control unit is configured to provide at least a single processor instruction to the vector computational unit. The single processor instruction specifies a plurality of component instructions to be executed by the vector computational unit (e.g., in response to the single processor instruction). For example, a control unit provides to the vector computational unit a single vector instruction, such as an instruction triad, that includes multiple component instructions. In some embodiments, an instruction triad is a simple processor instruction that includes up to three component instructions, such as a separate load instruction, arithmetic logic unit (ALU) instruction, and store instruction. The three component instructions are received and executed by the vector computational unit (e.g., in response to the instruction triad). For example, a vector computational unit receiving an instruction triad that bundles a load instruction, an ALU instruction, and a store instruction executes the load instruction, the arithmetic instruction, and the store instruction. In various embodiments, in response to the single processor instruction, the plurality of processing elements of the vector computational unit are configured to process different data elements in parallel with other processing elements. For example, each processing element is capable of processing in parallel a different data element from the input vector to the vector computational unit. As another example, each of the component instructions of a single vector processor instruction triad may be applied to each of the elements of a vector input to complete the processing of an entire input vector of N elements in parallel using the vector computational unit.

**[0022]** FIG. 1 is a block diagram illustrating an embodiment of a microprocessor system for performing machine learning processing. In the example shown, microprocessor system 100 includes control unit 101, data input 103, weight

input 105, matrix processor 107, vector engine 111, and post-processing unit 115. Data input 103 and weight input 105 are input modules for preparing data for matrix processor 107. In some embodiments, data input 103 and weight input 105 each include an input data formatter, a cache or buffer, and/or a logic circuit for preparing data for matrix processor 107. For example, data input 103 may prepare N operands from a two-dimensional array corresponding to image data and weight input 105 may prepare M operands corresponding to a vector of weight values to be processed by matrix processor 107. In some embodiments, the process of FIG. 5 is performed to prepare instructions for operating on microprocessor system 100, including matrix processor instructions for matrix processor 107 and vector engine instructions for vector engine 111. In some embodiments, microprocessor system 100, including vector engine 111, performs the processes described below with respect to FIGS. 6A, 6B, and 8.

[0023] In some embodiments, matrix processor 107 is a computational array that includes a plurality of computation units. For example, a matrix processor receiving M operands and N operands from weight input 105 and data input 103, respectively, includes M×N computation units. In the figure shown, the small squares inside matrix processor 107 depict that matrix processor 107 includes a logical two-dimensional array of computation units. Computation unit 109 is one of a plurality of computation units of matrix processor 107. In some embodiments, each computation unit is configured to receive one operand from data input 103 and one operand from weight input 105. In some embodiments, the computation units are configured according to a logical two-dimensional array but the matrix processor is not necessarily fabricated with computation units laid out as a physical two-dimensional array. For example, the i-th operand of data input 103 and the j-th operand of weight input 105 are configured to be processed by the i-th×j-th computation unit of matrix processor 107.

[0024] In various embodiments, the data width of components data input 103, weight input 105, matrix processor 107, vector engine 111, and post-processing unit 115 are wide data widths and include the ability to transfer more than one operand in parallel. In some embodiments, data input 103 and weight input 105 are each 96-bytes wide. In some embodiments, data input 103 is 192-bytes wide and weight input 105 is 96-bytes wide. In various embodiments, the width of data input 103 and weight input 105 is dynamically configurable. For example, data input 103 may be dynamically configured to 96 or 192 bytes and weight input 105 may be dynamically configured to 96 or 48 bytes. In some embodiments, the dynamic configuration is controlled by control unit 101. In various embodiments, a data width of 96 bytes allows 96 operands to be processed in parallel. For example, in an embodiment with data input 103 configured to be 96-bytes wide, data input 103 can transfer 96 operands to matrix processor 107 in parallel.

[0025] In various embodiments, matrix processor 107 is configured to receive N bytes from data input 103 and M bytes from weight input 105 and includes at least M×N computation units. For example, matrix processor 107 may be configured to receive 96 bytes from data input 103 and 96 bytes from weight input 105 and includes at least 96×96 computation units. As another example, matrix processor 107 may be configured to receive 192 bytes from data input 103 and 48 bytes from weight input 105 and includes at least

192×48 computation units. In various embodiments, the dimensions of matrix processor 107 may be dynamically configured. For example, the default dimensions of matrix processor 107 may be configured to receive 96 bytes from data input 103 and 96 bytes from weight input 105 but the input dimensions may be dynamically configured to 192 bytes and 48 bytes, respectively. In various embodiments, the output size of each computation unit is equal to or larger than the input size. For example, in some embodiments, the input to each computation unit is two 1-byte operands, one corresponding to an operand from data input 103 and one from weight input 105, and the output of processing the two operands is a 4-byte result. As another example, matrix processor 107 may be configured to receive 96 bytes from data input 103 and 96 bytes from weight input 105 and output 96 4-byte results. In some embodiments, the output of matrix processor 107 is a vector. For example, a matrix processor configured to receive two 96-wide input vectors, where each element (or operand) of the input vector is one byte in size, can output a 96-wide vector result where each element of the vector result is 4-bytes in size.

[0026] In various embodiments, each computation unit of matrix processor 107 is a sub-circuit that includes an arithmetic logic unit, an accumulator, and a shadow register. In the example shown, the computation units of matrix processor 107 can perform an arithmetic operation on the M operands and N operands from weight input 105 and data input 103, respectively. In various embodiments, each computation unit is configured to perform one or more multiply, add, accumulate, and/or shift operations. In some embodiments, each computation unit is configured to perform a dot-product operation. For example, in some embodiments, a computation unit may perform multiple dot-product component operations to calculate a dot-product result. For example, the array of computation units of matrix processor 107 may be utilized to perform convolution steps required for performing inference using a machine learning model. A two-dimensional data set, such as an image, may be formatted and fed into matrix processor 107 using data input 103, one vector at a time. In parallel, a vector of weights may be applied to the two-dimensional data set by formatting the weights and feeding them as a vector into matrix processor 107 using weight input 105. Corresponding computation units of matrix processor 107 perform a matrix processor instruction on the corresponding operands of the weight and data inputs in parallel.

[0027] In some embodiments, vector engine 111 is a vector computational unit that is communicatively coupled to matrix processor 107. Vector engine 111 includes a plurality of processing elements including processing element 113. In the figure shown, the small squares inside vector engine 111 depict that vector engine 111 includes a plurality of processing elements arranged as a vector. In some embodiments, the processing elements are arranged in a vector in the same direction as data input 103. In some embodiments, the processing elements are arranged in a vector in the same direction as weight input 105. In various embodiments, the data size of the processing elements of vector engine 111 is the same size or larger than the data size of the computation units of matrix processor 107. For example, in some embodiments, computation unit 109 receives two operands each 1 byte in size and outputs a result 4 bytes in size. Processing element 113 receives the 4-byte result from computation unit 109 as an input 4 bytes



in size. In various embodiments, the output of vector engine 111 is the same size as the input to vector engine 111. In some embodiments, the output of vector engine 111 is smaller in size compared to the input to vector engine 111. For example, vector engine 111 may receive up to 96 elements each 4 bytes in size and output 96 elements each 1 byte in size. In various embodiments, vector engine 111 performs quantization on the output result resulting in the output of vector engine 111 being smaller in size compared to the input to vector engine 111. In various embodiments, the quantization is performed as part of a single instruction. For example, a quantization and a non-linear function are performed as a single processor instruction. As described above, in some embodiments, the communication channel from data input 103 and weight input 105 to matrix processor 107 is 96-elements wide with each element 1 byte in size and matches the output size of vector engine 111 (96-elements wide with each element 1 byte in size).

**[0028]** In some embodiments, the processing elements of vector engine 111, including processing element 113, each include an arithmetic logic unit (ALU) (not shown). For example, in some embodiments, the ALU of each processing element is capable of performing arithmetic operations. In some embodiments, each ALU of the processing elements is capable of performing in parallel a rectified linear unit (ReLU) function and/or scaling functions. In some embodiments, each ALU is capable of performing a non-linear function including non-linear activation functions. In various embodiments, each processing element of vector engine 111 includes one or more flip-flops for receiving input operands. In some embodiments, each processing element has access to a slice of a vector engine accumulator and/or vector registers of vector engine 111. For example, a vector engine capable of receiving 96-elements includes a 96-element wide accumulator and one or more 96-element vector registers. Each processing element has access to a one-element slice of the accumulator and/or vector registers. In some embodiments, each element is 4-bytes in size. In various embodiments, the accumulator and/or vector registers are sized to fit at least the size of an input data vector. In some embodiments, vector engine 111 includes additional vector registers sized to fit the output of vector engine 111.

**[0029]** In some embodiments, the processing elements of vector engine 111 are configured to receive data from matrix processor 107 and each of the processing elements can process the received portion of data in parallel. As one example of a processing element, processing element 113 of vector engine 111 receives data from computation unit 109 of matrix processor 107. In various embodiments, vector engine 111 receives a single vector processor instruction and in turn each of the processing elements performs the processor instruction in parallel with the other processing elements. In some embodiments, the processor instruction includes one or more component instructions, such as a load, a store, and/or an arithmetic logic unit operation. In various embodiments, a no-op operation may be used to replace a component instruction.

**[0030]** In the example shown, the dotted arrows between data input 103 and matrix processor 107, weight input 105 and matrix processor 107, matrix processor 107 and vector engine 111, and vector engine 111 and post-processing unit 115 depict a coupling between the respective pair of components that is capable of sending multiple data elements such as a vector of data elements. As an example, the

communication channel between matrix processor 107 and vector engine 111 may be 96×32 bits wide and support transferring 96 elements in parallel where each element is 32 bits in size. As another example, the communication channel between vector engine 111 and post-processing unit 115 may be 96×1 byte wide and support transferring 96 elements in parallel where each element is 1 byte in size. In various embodiments, data input 103 and weight input 105 are coupled to a memory module (not shown in FIG. 1) and may each receive input data from the memory module. In some embodiments, vector engine 111 is additionally coupled to a memory module (not shown in FIG. 1) and may receive input data from the memory module in addition or alternatively to input from matrix processor 107. In the various embodiments, a memory module is typically a static random access memory (SRAM).

**[0031]** In some embodiments, one or more computation units of matrix processor 107 may be grouped together into a lane such that matrix processor 107 has multiple lanes. In various embodiments, the lanes of matrix processor 107 may be aligned with either data input 103 or weight input 105. For example, a lane aligned with weight input 105 includes a set of computation units that are configured to receive as input every operand of weight input 105. Similarly, a lane aligned with data input 103 includes a set of computation units that are configured to receive as input every operand of data input 103. In the example shown in FIG. 1, the lanes are aligned along weight input 105 in a vertical column and each lane feeds to a corresponding lane of vector engine 111. In some embodiments, each lane is a vertical column of sub-circuits that include multiply, add and/or accumulate, and shift functionality. In some embodiments, matrix processor 107 includes a matrix of tiles and each tile is a matrix of computation units. For example, a 96×96 matrix processor may include a matrix of 6×6 tiles, where each tile includes 16×16 computation units. In some embodiments, a vertical lane is a single column of tiles. In some embodiments, a horizontal lane is a single row of tiles. In various embodiments, the dimensions of the lane may be configured dynamically and may be utilized for performing alignment operations on the input to matrix processor 107, vector engine 111, and/or post-processing unit 115. In some embodiments, the dynamic configuration is performed by or using control unit 101 and/or with using processor instructions controlled by control unit 101.

**[0032]** In some embodiments, control unit 101 synchronizes the processing performed by matrix processor 107, vector engine 111, and post-processing unit 115. For example, control unit 101 may send processor specific instructions to each of matrix processor 107, vector engine 111, and post-processing unit 115. Control unit 101 may send matrix processor instructions to matrix processor 107. A matrix processor instruction may be a computational array instruction that instructs a computational array to perform an arithmetic operation, such as a dot-product or dot-product component, using specified operands from data input 103 and/or weight input 105. Control unit 101 may send vector processor instructions to vector engine 111. For example, a vector processor instruction may include a single processor instruction with a plurality of component instructions to be executed together by the vector computational unit. Control unit 101 may send post-processing instructions to post-processing unit 115. In various embodiments, control unit 101 synchronizes data that is fed to matrix processor 107

from data input 103 and weight input 105, to vector engine 111 from matrix processor 107, and to post-processing unit 115 from vector engine 111. In some embodiments, control unit 101 synchronizes the data between different components of microprocessor system 100 including between data input 103, weight input 105, matrix processor 107, vector engine 111, and/or post-processing unit 115 by utilizing processor specific memory, queue, and/or dequeue operations. In some embodiments, data and instruction synchronization is performed by control unit 101. In some embodiments, data and instruction synchronization is performed by control unit 101 that includes one or more sequencers to synchronize processing between matrix processor 107, vector engine 111, and/or post-processing unit 115.

[0033] In some embodiments, matrix processor 107 and vector engine 111 are utilized for processing convolution layers. In some embodiments, vector engine 111 is utilized for performing non-linear functions such as an activation function on the output of matrix processor 107. For example, matrix processor 107 may be used to calculate a dot-product and vector engine 111 may be used to perform an activation function such as a rectified linear unit (ReLU) or sigmoid function. In some embodiments, post-processing unit 115 is utilized for performing pooling operations. In some embodiments, post-processing unit 115 is utilized for formatting and storing the processed data to memory and may be utilized for synchronizing memory writing latency.

[0034] FIG. 2 is a block diagram illustrating an embodiment of a microprocessor system for performing machine learning processing. In the example shown, microprocessor system 200 includes control unit 201, vector input 203, vector engine input queue 207, vector engine 211, and post-processing unit 215. Vector engine input queue 207 includes a plurality of computation units including computation units 209 and 221-229 and vector engine 211 includes a plurality of processing elements including processing elements 213 and 231. Vector input 203 is an input module for feeding data into vector engine input queue 207. In some embodiments, vector input 203 includes an input data formatter, a cache or buffer, and/or a logic circuit for preparing data for vector engine input queue 207. For example, vector input 203 may prepare N operands from a two-dimensional array to be processed by vector engine 211 utilizing vector engine input queue 207 as a first-in-first-out (FIFO) input queue. In some embodiments, vector input 203 is coupled to memory (not shown in FIG. 2), such as static random access memory (SRAM) for retrieving data.

[0035] In various embodiments, control unit 201, vector input 203, vector engine input queue 207, vector engine 211, and post-processing unit 215 are, respectively, control unit 101, data input 103, matrix processor 107, vector engine 111, and post-processing unit 115 of FIG. 1. For example, matrix processor 107 of FIG. 1 may be used to implement an input queue such as vector engine input queue 207 by receiving data from data input 103 of FIG. 1 and repeatedly shifting each vector of input towards vector engine 111 of FIG. 1.

[0036] In some embodiments, vector engine input queue 207 is a computational array unit and includes a matrix of computation units whose columns are first-in-first-out (FIFO) queues. In the example shown, vector engine input queue 207 is an input queue for vector input 203 and functions as a wide first-in-first-out (FIFO) queue to feed multiple data elements from vector input 203 to vector

engine 211. For example, computation units 221-229 make up a vertical column of computation units that work together as a single FIFO queue. In various embodiments, vector engine input queue 207 includes multiple FIFO queues made up of vertical columns of computation units similar to computation units 221-229. For example, in an embodiment where vector engine input queue 207 is 96 computation units wide, vector engine input queue 207 has 96 vertical columns of computation units that correspond to 96 FIFO queues. As a further example, in an embodiment where vector engine input queue 207 is 96 computation units long, vector engine input queue 207 has FIFO queues that are 96 stages long.

[0037] In various embodiments, each first-in-first-out (FIFO) queue works in parallel and shifts input received from the vector input 203 along the FIFO queue to vector engine 211. The first row of computation units of vector engine input queue 207, which includes computation unit 221, is connected to the vector input 203. The first row of computation units is configured to receive an entire row of data from vector input 203 in parallel. The last row of computation units of vector engine input queue 207 is connected to the row of processing elements of vector engine 211. For example, the last row of computation units of vector engine input queue 207 includes computation units 229 and 209. Computation unit 209 is connected to processing element 213 and computation unit 229 is connected to processing element 231. Processing elements 213 and 231 are configured to receive the data output elements of computation units 209 and 229, respectively. The processing elements of vector engine 211 receive an entire row of data from the last row of computation units of vector engine input queue 207 in parallel. In various embodiments, when the last row of computation units of vector engine input queue 207 has data available to dequeue, a dequeue ready signal is received by vector engine 211 to indicate the vector engine input queue 207 is ready to receive a queue operation.

[0038] In the example described, the data from the first row of computation units is shifted down the column to the next row of computation units in the logical direction towards vector engine 211. For example, an input corresponding to a data element of vector input 203 is received as an operand at computation unit 221 and shifted from computation unit 221 to computation unit 222, from computation unit 222 to computation unit 223, from computation unit 223 to computation unit 224, and so forth, until an operand received at computation unit 221 is incrementally shifted from computation unit 221 to computation unit 229 via the intermediate computation units 222-228. In various embodiments, a data element pushed into the FIFO takes as many shifts as the FIFO is deep in computation units. For example, a FIFO queue with 96 computation units and 96 stages long requires 96 shifts to dequeue an inserted element. In various embodiments, each stage of the FIFO can shift an operand in parallel with the other stages. For example, while each intermediate computation unit in the FIFO queue shifts its operand to the next computation unit, the first computation unit can retrieve the next data element from vector input 203 and the last computation unit can dequeue its data element to be received by the corresponding processing element of vector engine 211. In the example described, each computation unit along each row of computation units works in parallel to shift its corresponding data element originally received from vector input 203 to vector engine 211.

[0039] In some embodiments, vector engine input queue 207 is coupled to vector input 203 and one dimension of the matrix of computation units matches the dimension of vector input 203. For example, in an embodiment with vector input 203 having a width of 96 bytes, vector engine input queue 207 has a matrix of computation units with a width of at least 96 bytes. In some embodiments, the width of vector input 203 and the corresponding width of the inputs to vector engine input queue 207 are dynamically configurable. For example, vector input 203 can be dynamically configured to 96 bytes or 96×2 bytes and the corresponding width of inputs to vector engine input queue 207 are configurable to 96 bytes or 96×2 bytes, respectively. In some embodiments, the configuration is performed using control unit 201 and/or processor instructions to vector engine input queue 207.

[0040] In some embodiments, vector engine 211 is a vector computational unit that is communicatively coupled to vector engine input queue 207. Vector engine 211 includes a plurality of processing elements including processing elements 213 and 231. In the figure shown, the small squares inside vector engine 211 depict that vector engine 211 includes a plurality of processing elements arranged as a vector. In some embodiments, the processing elements are arranged in a vector in the same direction as vector input 203. In various embodiments, the data size of the processing elements of vector engine 211 is the same size or larger than the data size of the computation units of vector engine input queue 207. For example, in some embodiments, computation unit 209 receives an operand 1 byte in size and dequeues an output to processing element 213 also having a size of 1 byte. Processing element 213 receives the 1 byte output from computation cell 209 as an input 1 byte in size. In various embodiments, the output of vector engine 211 is the same size as the input to vector engine 211. In various embodiments, the output of vector engine 211 is smaller in size as compared to the input to vector engine 211. For example, vector engine 211 may receive up to 96 elements each 4 bytes in size and output 96 elements each 1 byte in size. In some embodiments, the communication channel from vector input 203 to vector engine input queue 207 is 96 elements wide with each element 1 byte in size and matches the output size of vector engine 211 (96 elements wide with each element 1 byte in size).

[0041] In some embodiments, the processing elements of vector engine 211, including processing elements 213 and 231, each include an arithmetic logic unit (not shown) and are described in further detail with respect to vector engine 111 of FIG. 1. In some embodiments, the processing elements of vector engine 211 are configured to receive data from vector engine input queue 207 and each of the processing elements can process the received portion of data in parallel. As one example of a processing element, processing elements 213 and 231 of vector engine 211 receive data from computation units 209 and 229, respectively, of vector engine input queue 207. In various embodiments, vector engine 211 receives a single vector processor instruction and in turn each of the processing elements performs the processor instruction in parallel with the other processing elements. In some embodiments, the processor instruction includes one or more component instructions, such as a load, a store, and/or an arithmetic logic unit operation. In various embodiments, a no-op operation may be used to replace a component instruction.

[0042] In the example shown, the dotted arrows between vector input 203 and vector engine input queue 207, vector engine input queue 207 and vector engine 211, and vector engine 211 and post-processing unit 215 depict a coupling between the respective pair of components that is capable of sending multiple data elements. As an example, the communication channel between vector engine input queue 207 and vector engine 211 may be 96×32 bits wide and support transferring 96 elements in parallel where each element is 32 bits in size. As another example, the communication channel between vector engine 211 and post-processing unit 215 may be 96×1 byte wide and support transferring 96 elements in parallel where each element is 1 byte in size. In various embodiments, vector input 203 is coupled to a memory module (not shown in FIG. 2) and may receive input data from the memory module. In some embodiments, vector engine 211 is additionally coupled to a memory module (not shown in FIG. 1) and may receive input data from the memory module in addition or alternatively to input from vector engine input queue 207. In the various embodiments, a memory module is typically a static random access memory (SRAM).

[0043] In some embodiments, one or more computation units of vector engine input queue 207 may be grouped together into a vertical column such that vector engine input queue 207 has multiple vertical column lanes. In the example shown in FIG. 2, the lanes are aligned along the same vertical columns as the first-in-first-out (FIFO) queues described above and each lane feeds to a corresponding lane of vector engine 211. In some embodiments, each lane is a vertical column of sub-circuits that include multiply, add and/or accumulate, and shift functionality. In some embodiments, a vertical lane is a single column of computation units. In some embodiments, a vertical lane is a group of multiple columns of adjacent computation units. In various embodiments, the dimensions of the lane may be configured dynamically and may be utilized for performing alignment operations on the input to vector engine input queue 207, vector engine 211, and/or post-processing unit 215. In some embodiments, the dynamic configuration is performed by or using control unit 201 and/or with using processor instructions controlled by control unit 201.

[0044] In some embodiments, control unit 201 synchronizes the processing performed by vector engine input queue 207, vector engine 211, and/or post-processing unit 215. For example, control unit 201 may send processor specific instructions to each of vector engine input queue 207, vector engine 211, and post-processing unit 215. Control unit 201 may send vector engine input queue instructions to vector engine input queue 207. In some embodiments, vector engine input queue instructions are a subset of the matrix processor instructions that matrix processor 107 of FIG. 1 is capable of responding to and is described further with respect to FIG. 1. A vector engine input queue instruction may be a computational array instruction that instructs a computational array to perform a load operation, a shift operation, or other appropriate instruction for interfacing with an input queue. Control unit 201 may send vector processor instructions to vector engine 211. For example, a vector processor instruction may include a single processor instruction with a plurality of component instructions to be executed together by the vector computational unit. Control unit 201 may send post-processing instructions to post-processing unit 215. In various embodiments, control unit

**201** synchronizes data that is fed to vector engine input queue **207** from vector input **203**, to vector engine **211** from vector engine input queue **207**, and to post-processing unit **215** from vector engine **211**. In some embodiments, control unit **201** synchronizes the data between different components vector input **203**, vector engine input queue **207**, vector engine **211**, and/or post-processing unit **215** by utilizing processor specific memory, queue, and/or dequeue operations. The functionality of control unit **201** is described in further detail with respect to control unit **101** of FIG. 1.

**[0045]** In some embodiments, control unit **201** is utilized to configure the size and number of data elements to be received by vector engine input queue **207**, vector engine **211**, and/or post-processing unit **215**. For example, in some embodiments, control unit **201** may be utilized to configure the input to vector engine input queue **207** as 96 elements each of size 1 byte or other appropriate variations such as 48 elements each of size 2 bytes, 96 elements each of size 2 bytes, 192 elements each of size 4 bits, etc. In some embodiments, vector engine input queue **207** is able to output a data element with a size larger than it can receive by performing a sequence of load and logical shift operations. For example, a 4-byte input data element is loaded into vector engine input queue **207** by reading four sequential 1-byte portions of the 4-byte input data element and logically shifting each byte to the appropriate bit fields. As another example, in some embodiments, control unit **201** may be utilized to configure the input to vector engine **211** as 96 elements each of size 4 bytes, or other appropriate variations such as 96 elements each of size 1 byte, 48 elements each of size 2 bytes, etc.

**[0046]** In various embodiments, post-processing unit **215** is utilized to perform post-processing of output from vector engine **211**. The post-processing functionality of post-processing unit **215** is described in further detail with respect to post-processing unit **115** of FIG. 1.

**[0047]** FIG. 3 is a block diagram illustrating an embodiment of a microprocessor system for performing machine learning processing. In the example shown, microprocessor system **300** includes control unit **301**, memory **307**, vector engine **311**, and post-processing unit **315**. In various embodiments, memory **307** is typically a static random access memory (SRAM). In various embodiments, post-processing unit **315** received input data from vector engine **311** and is utilized to perform post-processing of output from vector engine **311**. The post-processing functionality of post-processing unit **315** is described in further detail with respect to post-processing unit **115** of FIG. 1.

**[0048]** The block diagram of FIG. 3 depicts a system architecture embodiment where vector engine **311** is coupled to memory **307** and may retrieve data directly from memory **307**. In various embodiments, the size of the communication channel between memory **307** and vector engine **311** may be configured to transfer multiple data elements in parallel from memory **307** to vector engine **311**. For example, in an embodiment where vector engine **311** is capable of receiving 96 elements each of 32 bits in size in parallel, the size of the communication channel between memory **307** and vector engine **311** is configured to transfer 96 elements each of 32 bits in size from memory **307** to vector engine **311** in parallel. In some embodiments, memory **307** includes a data formatter (not shown) which may include a data cache or buffer and/or a logic circuit for formatting data from memory prior to transfer to vector engine **311**. For example,

data elements of size 1 byte may be stored on word boundaries in memory **307** and the data formatter is utilized to format and/or mask the data to byte boundaries. In various embodiments, control unit **301**, vector engine **311**, and post-processing unit **315** are, respectively, control unit **101**, vector engine **111**, and post-processing unit **115** of FIG. 1. In various embodiments, vector engine **311** may be further coupled to a matrix processor (not shown) as described with respect to matrix processor **107** of FIG. 1.

**[0049]** In some embodiments, vector engine **311** is a vector computational unit that is communicatively coupled to memory **307**. Vector engine **311** includes a plurality of processing elements including processing element **313**. In the figure shown, the small squares inside vector engine **311** depict that vector engine **311** includes a plurality of processing elements arranged as a vector. In some embodiments, the processing elements of vector engine **311**, including processing element **313**, each include an arithmetic logic unit (not shown). The processing elements of vector engine **311** are configured to receive data from memory **307** and each of the processing elements can process the received portion of data in parallel. In various embodiments, vector engine **311** receives a single vector processor instruction and in turn each of the processing elements performs the processor instruction in parallel with the other processing elements. In some embodiments, the processor instruction includes one or more component instructions, such as a load, a store, and/or an arithmetic logic unit operation. The functionality of vector engine **311** is described in further detail with respect to vector engine **111** and **211** of FIGS. 1 and 2, respectively.

**[0050]** In some embodiments, control unit **301** synchronizes the processing performed by vector engine **311** and post-processing unit **315**, and access to memory **307**. For example, control unit **301** may send processor specific instructions to each of vector engine **311** and post-processing unit **315**. In some embodiments, control unit **301** may send vector processor instructions to vector engine **311**. For example, a vector processor instruction may include a single processor instruction with a plurality of component instructions to be executed together by the vector computational unit. In some embodiments, control unit **301** may send post-processing instructions to post-processing unit **315**. In various embodiments, control unit **301** synchronizes data that is received by vector engine **311** from memory **307** and received by post-processing unit **315** from vector engine **311**. In some embodiments, control unit **301** synchronizes the data between different components vector engine **311** and/or post-processing unit **315** by utilizing vector engine and/or post-processing unit processor specific operations. The functionality of control unit **301** is described in further detail with respect to control unit **101** of FIG. 1.

**[0051]** In some embodiments, control unit **301** is utilized to configure the size and number of data elements to be received by vector engine **311** and/or post-processing unit **315**. For example, in some embodiments, control unit **301** may be utilized to configure vector engine **311** to receive 96 data elements each of size 4 bytes, or other appropriate variations such as 96 elements each of size 1 byte, 48 elements each of size 2 bytes, etc. As described further with respect to FIGS. 1 and 2, the dotted arrows between vector engine **311** and post-processing unit **315** depict a coupling between the respective pair of components that is capable of sending multiple data elements. As an example, the com-

munication channel between vector engine 311 and post-processing unit 315 may be 96×1 byte wide and support transferring 96 elements in parallel where each element is 1 byte in size.

**[0052]** FIG. 4A is a block diagram illustrating an embodiment of a vector computational unit for performing machine learning processing. In the example shown, microprocessor system 400 includes vector computational unit 401, input bus 411, and output bus 431. Input to vector computational unit 401 arrives from input bus 411. Output from vector computational unit 401 is written to output bus 431. In some embodiments, input bus 411 and output bus 431 are a single bus that includes the functionality of both input bus 411 and output bus 431. In various embodiments, input bus 411 and output bus 431 are wide data buses that allow the transfer of multiple data elements in parallel. For example, input bus 411 may be 96×32 bits wide and output bus 431 may be 96 bytes wide to accommodate the parallel processing functionality of computational unit 401. In some embodiments, vector computational unit 401 receives vector computational unit instructions via input bus 411. In some embodiments, vector computational unit 401 receives vector computational unit instructions via a communication channel other than input bus 411 such as an instruction bus (not shown).

**[0053]** In various embodiments, vector computational unit 401 is vector engine 111, 211, and/or 311 of FIGS. 1, 2, and 3, respectively. In some embodiments, input bus 411 is connected to matrix processor 107 of FIG. 1, vector engine input queue 207 of FIG. 2, and/or memory 307 of FIG. 3. In some embodiments, output bus 431 is connected to post-processing units 115, 215, and/or 315 of FIGS. 1, 2, and 3, respectively. In various embodiments, vector computational unit 401 is bi-directionally coupled to a control unit (not shown) of microprocessor system 400 external to vector computational unit 401, such as control units 101, 201, and/or 301 of FIGS. 1, 2, and 3, respectively. In various embodiments, the control unit of microprocessor system 400 sends vector computational unit instructions to vector computational unit 401. In some embodiments, the control unit of microprocessor system 400 includes one or more sequencers for synchronizing instructions and data to vector computational unit 401.

**[0054]** In the example shown, vector computational unit 401 includes registers 421, vector engine control logic 423, input buffer 425, arithmetic logic units (ALUs) 427, and output buffer 429. Input data from input bus 411 is received by input buffer 425 and output written to output bus 431 is written from output buffer 429. In some embodiments, input buffer 425 and output buffer 429 are data buffers or caches and provide memory synchronization functionality. For example, in some embodiments, input reads from input bus 411 and/or output writes to output bus 431 have an unpredictable latency that can be smoothed out by utilizing input buffer 425 to receive input data and output buffer 429 for storing calculated results. As another example, output bus 431 may not be available when output from ALUs 427 is ready for writing. In some embodiments, output buffer 429 allows ALUs 427 to continue processing pending data until output bus 431 is available for writing the results stored at output buffer 429. In various embodiments, input bus 411 and output bus 431 are communication channels controlled by a control unit (not shown) of microprocessor system 400.

**[0055]** As described above, in various embodiments, a vector computational unit includes a plurality of processing

elements. In some embodiments, each processing element includes individual functionality for loading data, storing data, and performing arithmetic logic unit operations. The individual processing elements are not depicted in the block diagram of FIG. 4A. In various embodiments, arithmetic logic units (ALUs) 427 include the corresponding arithmetic logic unit (ALU) of each processing unit. Similarly, input buffer 425 and output buffer 429 include corresponding input buffers and output buffers for each processing unit. In various embodiments, ALUs 427 include ALU logic for processing every element of an input vector to vector computational unit 401 in parallel. In some embodiments, ALUs 427 include logic for quantizing the ALU result. In various embodiments, the ALU logic, for example, logic for performing a non-linear function and quantization, can be performed in response to a single processor instruction.

**[0056]** In various embodiments, registers 421 includes registers for implementing the functionality of vector computational unit 401. For example, registers 421 may be used to store operands for performing vector computational unit instructions, to implement bit masks, and to reference vector elements using different memory-sized register aliases, among other appropriate functionality. In some embodiments, registers 421 include arithmetic instruction vector registers; mask registers; registers for performing arithmetic operations such as add, subtract, and floating point operations; and/or registers for aliasing vector elements. In some embodiments, the registers used for aliasing vector elements are also utilized for performing arithmetic operations.

**[0057]** In some embodiments, registers 421 include arithmetic instruction vector registers. For example, registers may be used as operands for load operations, store operations, and arithmetic logic unit (ALU) operations. As another example, in some embodiments, an ALU operation may take as arguments up to four vector registers, three as source registers and one as a destination register. In various embodiments, the vector registers used by processor operations are aliased to different vector elements based on the size of the vector element. For example, in some embodiments, a different set of vector registers are available for operating on 8-bit, 16-bit, 32-bit, and/or floating point values. In some embodiments, the set of vector registers for 32-bit values is also used for floating point values. In various embodiments, 32-bit vector registers are aliased to 16-bit vector registers and 8-bit vector registers. For example, one 32-bit vector register is aliased to two 16-bit vector registers and four 8-bit vector registers. As another example, a vector computational unit 401 with eight 96×32-bit vector registers (registers RD0-RD7) is aliased to sixteen 96×16-bit vector registers (registers RW0-RW15), and thirty-two 96×8-bit vector registers (registers RB0-RB31). RD0 is a 96×32-bit vector register, RW0 is a 96×16-bit vector register, and RB0 is a 96×8-bit vector register. A further example of vector register aliasing is depicted in FIG. 4B.

**[0058]** In some embodiments, registers 421 include one or more bit mask registers based on the number of processing elements of vector computational unit 401. For example, a vector computational unit with 96 processing elements may include one or more 96-bit mask registers. In various embodiments, a mask register may be set by loading a bit-mask from memory. A mask register may be used to store the results of logical operations performed on input data to vector computational unit 401.

[0059] In some embodiments, registers 421 include registers for performing arithmetic operations such as add, subtract, and floating point operations. For example, in some embodiments, vector computational unit 401 includes registers for storing carry-out bits for vector add and subtract instructions and status bits corresponding to floating point instructions.

[0060] In some embodiments, vector computational unit 401 includes an instruction buffer (not shown) for storing a sequence of vector computational unit instructions. In some embodiments, the instruction buffer is a command queue. In various embodiments, the instruction buffer includes one or more pointers to reference the current and/or last instruction to be performed. In various embodiments, the instruction buffer acts as a cache of vector computational unit instructions. For example, one or more vector computational unit instructions are loaded into an instruction buffer of vector computational unit 401 and cached until the instructions can be executed. As instructions are executed and no longer needed, new instructions may be loaded into the instruction buffer. In some embodiments, the vector computational unit instructions are received from an external instruction command queue via a control logic (not shown) of microprocessor system 400.

[0061] In some embodiments, vector computational unit 401 includes a vector engine control logic 423. Vector engine control logic 423 is utilized to implement the functionality of the vector computational unit 401 including fetching vector computational unit instructions, decoding the instructions, and/or executing the instructions. In various embodiments, the vector engine control logic 423 includes logic for reading, writing, masking, and/or aliasing the data via input buffer 425, output buffer 429, and registers 421. In some embodiments, vector computational unit 401 receives a dequeue ready signal and determines using vector engine control logic 423 that data is available via input bus 411. For example, vector engine control logic 423 may dequeue data from an input first-in-first-out queue (not shown) attached to input bus 411 on receipt of a dequeue ready signal.

[0062] FIG. 4B is a table illustrating an exemplary aliasing of vector registers. Table 450 illustrates the aliasing of vector registers for a vector computational unit embodiment with eight 96×32-bit vector registers (registers RD0-RD7) aliased to sixteen 96×16-bit vector registers (registers RW0-RW15), and thirty-two 96×8-bit vector registers (registers RB0-RB31). In some embodiments, the vector registers in Table 450 are the vector registers of registers 421 of vector computational unit 401 of FIG. 4A. In the example shown, row 451 includes columns for the bytes 0, 1, 2, and 3 that are aliased to the respective registers listed in the rows below it. Rows 453, 463, and 473 correspond to 96×32-bit vector registers RD0, RD1, and RD7. Rows 455, 465, and 475 correspond to 96×16-bit vector registers RW0-3 and RW14-15. Rows 457, 467, and 477 correspond to 96×8-bit vector registers RB0-7 and RB28-31. In the example, bytes 0-3 are one of the 96 lanes of a vector computational unit such as vector engine 111, 211, and/or 311 of FIGS. 1, 2, and 3, respectively.

[0063] In the example shown, table 450 illustrates vector register aliasing for a single lane of the 96 lanes of a vector computational unit embodiment. The 96×32-bit vector register RD0 utilizes four bytes ordered from byte 0 to byte 3. The 96×16-bit vector registers RW0 and RW1 are aliased to 2 bytes each. Vector register RW0 is aliased to byte 0 and

byte 1 and vector register RW1 is aliased to byte 2 and byte 3. The 96×8-bit vector registers RB0-RB3 are aliased to 1 byte each corresponding to bytes 0-3, respectively. Similarly, the 96×32-bit vector register RD1 is aliased to the 96×16-bit vector registers RW2 (bytes 0 and 1) and RW3 (bytes 2 and 3), and the 96×8-bit vector registers RB4-RB7 for bytes 0-3, respectively. As another example, the 96×32-bit vector register RD7 is aliased to the 96×16-bit vector registers RW14 (bytes 0 and 1) and RW15 (bytes 2 and 3), and the 96×8-bit vector registers RB28-RB31 for bytes 0-3, respectively.

[0064] In various embodiments, vector computational unit instructions operate on all 96 lanes of a vector register in parallel. For example, for each of the 96 lanes, vector register RB0 operates on byte 0, vector register RB5 operates on byte 1, vector register RW2 operates on bytes 0 and 1, vector register RW15 operates on bytes 2 and 3, and vector register RD7 operates on bytes 0-3 in parallel.

[0065] FIG. 5 is a flow diagram illustrating an embodiment of a process for determining processor instructions for a microprocessor system. In some embodiments, the process of FIG. 5 converts a software program written with a high level programming language into a sequence of computational array and vector computational unit instructions for a microprocessor system with a computational array and a vector computational unit. In various embodiments, the microprocessor system is microprocessor system 100 of FIG. 1, a computational array is matrix processor 107 of FIG. 1, and a vector computational unit is vector engine 111 of FIG. 1. In various embodiments, the process of FIG. 5 is utilized to implement applications relying on machine learning including applications that perform inference using a machine learning model such as self-driving and driver-assisted automobiles.

[0066] At 501, a determination is made on the processing to be performed and the subset of processing to be assigned to different co-processing components such as a computational array, a vector computational unit, and/or a post-processing unit. In various embodiments, the processing is assigned based on the functionality and efficiency of the different co-processing components. For example, certain matrix-related operations are assigned to a computational array and operations involving non-linear functions such as activation functions may be assigned to a vector computational unit. In some embodiments, pooling operations are assigned to a post-processing unit. As another example, in some embodiments, at 501, a determination is made that a convolution operation requires a dot-product operation and that the dot-product operation best utilizes matrix processing performed by a computational array. In some embodiments, this determination is performed by compiling a machine learning application to target the microprocessor system described herein.

[0067] At 503, one or more matrix processor instructions are determined that correspond to the processing determined and assigned at 501. For example, the dot-product operation determined at 501 to be performed by a matrix processor is converted to one or more matrix processor instructions. In various embodiments, the matrix processor instructions are computational array instructions. As an example, the computational array instructions may require that one or more data vectors are received from a data input component, such as data input 103 of FIG. 1, and one or more weight vectors are received from a corresponding weight input component,

such as weight input **105** of FIG. 1. Additional computational array instructions may include the multiply, accumulate, and shift operations for processing a dot-product operation. For example, one or more dot-product component operations may be used to calculate a dot-product result. In various embodiments, the computational array instructions are directed to processing performed on received input data by the corresponding computation units of the computational array. In some embodiments, additional computational array instructions include instructions for preparing the dot-product result for processing by the vector computational unit.

**[0068]** At **505**, a determination is made regarding the vector engine instructions to be performed by the vector computational unit. For example, operations related to an activation function determined at **501** to be performed by a vector engine are converted to one or more vector engine instructions. In various embodiments, the vector engine instructions are vector computational unit instructions. As an example, the vector computational unit instructions may require that one or more data vectors are received from a computational array, such as matrix processor **107** of FIG. 1. Additional vector computational unit instructions may include operations for performing a non-linear activation function, such as a rectified linear unit (ReLU) function. In various embodiments, the vector computational unit instructions are directed to processing performed on received input data by the corresponding processing elements of the vector computational unit. In some embodiments, additional vector computational unit instructions include instructions for preparing the result of the processing elements for post-processing by the post-processing unit.

**[0069]** In various embodiments, each vector computational unit instruction is a single processor instruction that specifies a plurality of component instructions to be executed together by the vector computational unit. The execution of the plurality of component instructions is performed by the processing elements of the vector computational unit in parallel on different data input elements in response to a single vector computational unit instruction. For example, in some embodiments, a single processor instruction includes three component instructions: a separate load, arithmetic logic unit, and store instruction. The three component instructions are received and executed by the vector computational unit. In some embodiments, the bundling of component instructions into a single processing instruction is performed at **505**. In various embodiments, the order and selection of component instructions for bundling into a vector computational unit instruction is based on determined data hazards.

**[0070]** At **507**, a determination is made regarding the post-processing instructions to be performed by the post-processing unit. For example, operations related to post-processing functionality are determined at **501** to be performed by a post-processing unit and are converted to one or more post-processing instructions. As an example, the post-processing instructions may require that one or more data vectors are received from a vector computational unit, such as vector engine **111** of FIG. 1. Additional post-processing instructions may include operations for performing pooling layer functionality, such as a maxpooling. In various embodiments, post-processing instructions may include instructions for configuring the pooling functionality such as kernel size, stride, and/or spatial extent, among others. In

some embodiments, additional post-processing instructions include instructions for preparing and writing out the results of post-processing.

**[0071]** At **509**, the sequence corresponding to the execution of the collection of co-processor instructions determined at **503**, **505**, and **507** is scheduled. For example, the relative order and/or sequence of the respective processor instructions for the various co-processors, such as computational array, a vector computational unit, and/or a post-processing unit, is determined. In some embodiments, the sequence depends on the interaction and dependencies between the co-processors. For example, the input to a vector computational unit may depend on the availability of output results from a computational array. In various embodiments, dependencies including data hazards are determined and accounted for. For example, in various embodiments, vector computational unit instructions include a plurality of component instructions and can be executed such that multiple vector computational unit instructions are executed in parallel. Data hazards based on unavailable data resources are determined and accounted for. For example, no-ops may be inserted into the component instructions of a vector computational unit instruction to allow a load operation to complete before an arithmetic logic unit operation that depends on the completion of the load operation is performed. In some embodiments, the bundling of component instructions into a single vector computational unit instruction is determined at **509**. In some embodiments, some or all of the instruction scheduling, such as the ordering of co-processor instructions, is performed at **503** and **505** for a matrix processor and vector engine, respectively. For example, in some embodiments, the bundling of component instructions for each single vector computational unit instruction is determined at **505**.

**[0072]** In some embodiments, a control unit and/or one or more sequencers of a microprocessor system are utilized to initiate and coordinate the processing of the collection of co-processor instructions. For example, the instruction sequence determined at **509** is utilized by a control unit, such as control unit **101** of FIG. 1, and/or by one or more sequencers to issue the corresponding co-processor instructions to a computational array such as matrix processor **107** of FIG. 1, a vector computational unit such as vector engine **111** of FIG. 1, and/or a post-processing unit such as post-processing unit **113** of FIG. 1. In some embodiments, the functionality of one or more sequencers is performed by a control unit. For example, in some embodiments, the control unit includes an execute sequencer, memory access sequencers, network sequencers, and/or vector engine sequencers, among others.

**[0073]** FIG. 6A is a flow diagram illustrating an embodiment of a process for the running execution of a vector computational unit. The process of FIG. 6A may be performed by a vector computational unit to process elements of a vector in parallel. In various embodiments, a vector computational unit is vector engine **111**, **211**, **311**, and/or vector computational unit **401** of FIGS. 1, 2, 3, and 4A, respectively. In some embodiments, the process of FIG. 6A is initiated by a control unit such as control unit **101** of FIG. 1. In various embodiments, the transition between the steps of the process in FIG. 6A is performed by a control logic of the vector computational unit such as vector engine control logic **423** of FIG. 4A.

[0074] At 601, a vector engine instruction is retrieved. In various embodiments, a vector engine instruction is a vector computational unit instruction and specifies a plurality of component instructions. For example, an instruction triad is a single vector computational unit instruction specifying up to three component instructions. An example instruction triad includes a load operation, an arithmetic logic unit operation, and a store operation as a single instruction. At 601, once the instruction is retrieved, the process continues to both 603 and 605.

[0075] At 603, a determination is made as to whether additional instructions are pending. For example, the next vector engine instruction may be available and ready for retrieving. As another example, an instruction buffer for caching pending instructions may be empty and requires retrieving and/or waiting for the next available instruction. In some embodiments, the availability of additional instructions is based on inspecting a pointer referencing the last valid instruction in the instruction buffer. Processing proceeds to step 609 in response to no available additional instructions. Processing proceeds back to 601 in response to the availability of one or more additional instructions.

[0076] At 605, the vector engine instruction retrieved at 601 is decoded. In various embodiments, a single vector engine instruction specifies one or more component instructions. In various embodiments, the instruction and the component instructions are decoded. For example, an instruction triad containing a load, an arithmetic logic unit, and a store component instruction is decoded into the separate component operations. In some embodiments, the decoding determines both the opcode and the arguments corresponding to the opcode for each component operation. As one example, a load component instruction contains both the opcode corresponding to a byte vector dequeue operation and the corresponding destination vector register to store the vector of bytes as a result of the dequeue. As another example, an add component instruction contains both the opcode corresponding to a signed 16-bit add operation and the corresponding vector registers for the source and destination arguments.

[0077] At 607, the instruction decoded at 605 is executed. In some embodiments, a single vector engine instruction, which specifies multiple component instructions, is executed by the processing elements of the vector computational unit. For example, a vector of processing elements executes the single vector engine instruction decoded at 605. In some embodiments, each of the component instructions of the single vector engine instruction is further executed in parallel by each of the processing elements. For example, for each processing element, a load instruction and an arithmetic logic unit instruction may be executed in parallel. In some embodiments, a load instruction, an arithmetic logic unit instruction, and a store instruction may be executed in parallel. For example, the following component operations are performed in parallel by each processing cell of the vector engine: a vector of input data is loaded from an input accumulator into a vector register, a floating point multiply operation is performed on two different vector registers by an arithmetic logic unit (ALU), and a vector of 16-bit elements is stored from a vector register to memory. In various embodiments, once the processing elements have finished execution of component instructions, the processing for the vector engine instruction is complete.

[0078] At 609, the vector computational unit waits for the next instruction. For example, the vector computational unit waits until an instruction buffer for caching pending instructions contains a valid instruction to be executed. As another example, the vector computational unit waits until the next instruction is received from memory and made available to the vector computational unit. In some embodiments, the vector computational unit halts at 609 pending the availability of an additional instruction. In various embodiments, the vector computational unit may respond to interrupts at 609 while waiting for an additional instruction. In response to the arrival of an additional instruction, processing continues back to 601.

[0079] FIG. 6B is a flow diagram illustrating an embodiment of a process for processing vector data by a vector computational unit. For example, FIG. 6B illustrates the process applied to vector data received by a vector computational unit from an input source such as a computational array and/or a first-in-first-out (FIFO) queue. In some embodiments, the process of FIG. 6B illustrates the steps performed by a vector computational unit for performing a vector operation on a vector input to compute a vector result. In various embodiments, the process of FIG. 6B utilizes a plurality of processing elements of a vector computational unit to perform processing on elements of a vector in parallel. In various embodiments, vector computational unit is vector engine 111, 211, 311, and/or vector computational unit 401 of FIGS. 1, 2, 3, and 4A, respectively.

[0080] At 651, a load operation is decoded and issued. In some embodiments, a load operation is required to receive data into a vector computational unit. For example, in some embodiments, a dequeue operation is a load operation that dequeues a vector of data elements from a computational array to be received by the processing elements of the vector computational unit. In various embodiments, the load operation may be one of multiple component instructions that make up a single vector computational unit instruction. The decoding of the load operation determines the specific type of load operation and the appropriate operations. For example, various load operations exist to load different sized vector elements into different specified vector registers. At 651, the load operation is decoded and issued to initiate the receiving of input data such as the dequeuing of a vector of data results from a first-in-first-out (FIFO) queue.

[0081] At 653, the vector computational unit receives input data in the form of a vector as a result of the load operation issued at 651. For example, the vector computational unit receives a vector of input data elements from a computational array, such as matrix processor 107 of FIG. 1, a first-in-first-out (FIFO) queue, such as vector engine input queue 207 of FIG. 2, or other appropriate data source. In some embodiments, the input data is stored in an input buffer. In some embodiments, the input buffer utilizes a set of flip-flops and/or one or more accumulators to store the input data. An input buffer the size of the input vector may be utilized to store the input data so that it can be loaded into one or more vector registers at step 655.

[0082] At 655, vector data received at 653 is loaded into the appropriate registers. For example, the vector data read at 653 is loaded into the vector registers designated by the load instruction. In some embodiments, register aliasing is used to determine how data is loaded into a vector register. For example, data may be loaded into the same register's memory location but aligned to byte, half-word, or word



boundaries based on the instruction and aliased registers utilized. In some embodiments, the loading of vector data into vector registers utilizes a bit mask, such as a vector bit mask, to determine which bytes of a vector to load into which register memory locations. For example, a 96-bit mask may be utilized to determine which elements of a vector register should receive data.

**[0083]** At **657**, a determination is made on whether additional data is needed. For example, based on the current vector computational unit instruction, additional data may be needed before performing an arithmetic logic unit (ALU) operation. In response to not needing additional data, processing continues to **661**. As an example, processing continues to **661** in the event the current vector computational unit instruction includes an ALU component operation (such as an add operation) that is not a no-op operation. In response to needing additional data, for example, a load operation is pending and no ALU operation is pending, processing continues to **659**. In some embodiments, an instruction triad may replace an ALU operation with a no-op indicating that an ALU operation should not be performed for the current instruction.

**[0084]** At **659**, additional data is loaded into the vector computational unit for processing. For example, additional input data, such as a vector of input weights, may be loaded by reading memory, receiving the result of a matrix processor, dequeuing a first-in-first-out (FIFO) queue, or other appropriate technique. In some embodiments, additional data may be loaded by reading a memory such as a static random access memory (SRAM). In various embodiments, additional components such as a read buffer may be utilized to synchronize the loading of data and/or to account for read delays and latency. In various embodiments, the data loaded at **659** may be a vector of input data, such as a vector of weight inputs.

**[0085]** At **661**, a vector arithmetic logic unit (ALU) operation is performed. In various embodiments, vector ALU operations include vector operations for add (signed and unsigned), subtract (signed and unsigned), multiply, absolute value, and logical operators, among others. Vector ALU operations may be performed on different operand sizes. Example operand sizes include 8-bit, 16-bit, 32-bit, and floating point values. In some embodiments, the different operand sizes are determined based on register aliasing and/or the opcode of the operation. For example, a vector add operation on 8-bit operands utilizes 8-bit vector registers. As explained in more detail with respect to FIGS. 4A and 4B, register aliasing allows the same memory location to be referenced using different aliases. For example, a 32-bit block of memory can be referenced as a single 4-byte operand, two 2-byte operands, or four 1-byte operands depending on the desired result. In various embodiments, each processing element of the vector computational unit performs the same ALU operation (e.g., add, subtract, multiply, etc.) in parallel with the other processing elements. In some embodiments, the output result is a quantized version of the ALU result. For example, the output result is a quantized version that requires fewer bits to represent than the ALU result. In some embodiments, the ALU result is calculated using a result represented using fewer bits than the input operands. For example, input operands may be 4-bytes each and an output result may be 1-byte in size.

**[0086]** At **663**, the vector result of the arithmetic logic unit (ALU) operation performed at **661** is written out of the

vector computational unit. In some embodiments, the vector result is written out utilizing an output buffer that allows processing to continue for the next ALU operation in the event the output bus is not available to receive data. In some embodiments, the vector output result is transferred to a post-processing unit such as post-processing units **115**, **215**, and/or **315** of FIGS. 1, 2, and 3, respectively. For example, the result of performing an ALU operation is written to a post-processing unit for performing post-processing pooling operations. In some embodiments, the output vector result is written to memory such as static random access memory (SRAM). In various embodiments, the output is written out as a vector of elements such as a 96-element vector with each element having the size of 1 byte.

**[0087]** FIG. 7 is a block diagram illustrating an embodiment of an encoding format for a vector computational unit instruction. In the example shown, vector computational unit instruction **710** depicts the encoding of multiple component instructions specified by a single instruction. Vector computational unit instruction **740** further details the format of each of the multiple component instructions specified by a single instruction. Vector computational unit instruction **710** is an encoded instruction triad and includes load operation **711**, arithmetic logic unit (ALU) operation **713**, and store operation **715**. Vector computational unit instruction **740** includes fields: opcode **741**, register **743**, opcode **751**, registers **753**, opcode configuration field **755**, immediate field **757**, opcode **761**, and register **763**. The fields for component instructions (corresponding to a load operation, ALU operation, and store operation) depicted by vector computational unit instruction **710** map to vector computational unit instruction **740**. Vector computational unit instruction **740** includes an encoded load operation (opcode **741** and register **743**), arithmetic logic unit operation (opcode **751**, registers **753**, opcode configuration field **755**, and immediate field **757**), and store operation (opcode **761** and register **763**).

**[0088]** In some embodiments, a vector computational unit instruction is an instruction triad specifying three component instructions. For example, a load operation, arithmetic logic unit (ALU) operation, and store operation may be bundled into a single instruction using a 128-bit format. In various embodiments, a larger or smaller bit format may be utilized to bundle the three component instructions as appropriate. In some embodiments, load and store operations are encoded into 13 bits and ALU operations are encoded into 64 bits. In various embodiments, any remaining bits not used by the bundled load, store, and ALU operations are padding bits. In some embodiments, opcodes are encoded into 8 bits, registers are encoded into 5 bits, and immediate fields are encoded into 32 bits. In various embodiments, different length encodings may be utilized as appropriate and are based on the instruction size, number of supported vector operations, number of registers, vector size, and/or other appropriate factors. In some scenarios, a no-op operation is used when one or more of the component instructions are not utilized.

**[0089]** In the example shown, the encoded load operation of vector computational unit instruction **740** includes opcode **741** and register **743**. Opcode **741** corresponds to a vector load operation and register **743** is the corresponding destination vector register for the load operation. For example, opcode **741** may be used to store the opcode for a dequeue operation that loads data and register **743** is the destination register for storing the loaded data. In various embodiments,

the load operation is used to load a vector of input data into a vector register for processing by a vector computational unit. In some embodiments, opcode **741** is an 8-bit field and register **743** is a 5-bit field.

[0090] In the example shown, the encoded store operation of vector computational unit instruction **740** includes opcode **761** and register **763**. Opcode **761** corresponds to a vector store operation and register **763** is the corresponding source vector register for which the store operation should read a vector of data from. For example, opcode **761** may be used to store the opcode for a store operation that stores data from register **763** to external memory such as static random access memory (SRAM). In some embodiments, the start address of the memory used for storing is maintained by an external sequencer or control unit using a write pointer to reference a memory location. In some embodiments, the store operation is used to write a vector of data to an output data bus. In some embodiments, opcode **761** is an 8-bit field and register **763** is a 5-bit field.

[0091] In the example shown, the encoded arithmetic logic unit (ALU) operation includes opcode **751**, registers **753**, opcode configuration field **755**, and immediate field **757**. Opcode **751** is used to encode an ALU opcode. For example, ALU opcodes may include opcodes that correspond to vector operations for add (signed and unsigned), subtract (signed and unsigned), multiply, absolute value, and logical operators, among others. Depending on the vector ALU operation, the operation may utilize fields: registers **753**, opcode configuration field **755**, and immediate field **757**. In some embodiments, registers **753** specifies up to four vector registers including three source registers and one destination register. In some embodiments, registers **753** is a 20-bit field and utilizes 5 bits for each register.

[0092] In some embodiments, an encoded arithmetic logic unit (ALU) operation includes opcode configuration field **755** that is utilized by certain ALU operations. In some embodiments, opcode configuration field **755** is a 5-bit field and includes a register size field (2-bits), a mask bit (1-bit), and an immediate valid bit (1-bit). For example, in some scenarios, the value stored in the register size field (2-bits) may be used to specify the size of the registers (e.g., 8-bits, 16-bits, or 32-bits). As additional examples, a mask bit (1-bit) may be utilized to process immediate field **757** as a bit mask and an immediate valid bit (1-bit) may be utilized to identify the validity of immediate field **757**. In various embodiments, immediate field **757** is a 32-bit field that is utilized for ALU operations that require an immediate field. For example, a vector move operation may be configured to move a 32-bit value from immediate field **757** to a destination vector register.

[0093] In some embodiments, a vector computational unit supports a vector mask move instruction (not shown) to load a vector bit mask into a vector mask register. In some embodiments, a vector mask move instruction includes a corresponding opcode field, a destination register field, and an immediate field. As an example, the vector mask move loads a vector bit mask stored in the immediate field to the vector mask register. In some embodiments, the size of the vectors (e.g., 96 elements wide) supported by the vector computational unit requires a large enough immediate field (e.g., 96-bits) to store the bit mask. In some embodiments, the vector mask move instruction is not restricted to the encoding formats of vector computational unit instructions **710** and **740**. For example, based on the size of the im-

mediate field, the vector mask move may not be bundled with other component instructions.

[0094] In various embodiments, the component instructions of vector computational unit instructions are bundled together using the process of FIG. 5. In some embodiments, the encoding format of FIG. 7 is utilized by a vector computational unit such as vector engine **111**, **211**, **311**, and/or vector computational unit **401** of FIGS. 1, 2, 3, and 4A, respectively. In some embodiments, a vector computational unit instruction is issued to a vector computational unit by a sequencer of a microprocessor system or control unit containing a sequencer.

[0095] FIG. 8 is a flow diagram illustrating an embodiment of a process for performing a single vector computational unit instruction by a vector computational unit. The process of FIG. 8 may be performed by a vector computational unit on elements of a vector in parallel utilizing the processing elements of a vector computational unit. In some embodiments, the process of FIG. 8 is performed by a vector computational unit such as vector engine **111**, **211**, **311**, and/or vector computational unit **401** of FIGS. 1, 2, 3, and 4A, respectively.

[0096] At **801**, a vector computational unit instruction is fetched. In some embodiments, the instruction is fetched from an instruction buffer and/or command queue. In various embodiments, the instruction buffer includes one or more pointers to reference the current instruction to be performed. In various embodiments, the instruction buffer acts as a cache of vector computational unit instructions.

[0097] At **821**, the vector computational unit instruction is decoded. For example, a vector computational unit instruction that is an instruction triad is decoded into its three component instructions. In various embodiments, the arguments and fields utilized by each component instruction are decoded. For example, vector registers specified by a registers field, such as registers **753** of FIG. 7, are decoded into source and destination registers.

[0098] At **831**, the component instructions are issued. In some embodiments, the issuing of component instructions includes determining whether a resource and/or data hazards are present. In the event hazards are present, in some embodiments, the vector computational unit waits for the hazard to be resolved. For example, in the event of a resource hazard caused by a load operation in the previous clock cycle, the vector computational unit waits one or more clock cycles for the load to complete and for the resource to be available.

[0099] In some embodiments, the multiple component instructions are issued together and executed in parallel. For example, the load operation, arithmetic logic unit (ALU) operation, and store operation of an instruction triad are executed together and during the same clock cycle. In the scenario where the component instructions are executed together, each of the steps corresponding to executing a load operation (step **845**), an ALU operation (step **855**), and a store operation (step **865**) along with corresponding no-op alternatives (steps **843**, **854**, and **863**) are initiated in the same clock cycle and execution proceeds in parallel.

[0100] In some embodiments, the different component instructions are executed with staggered starts. For example, in some embodiments, the load operation is executed first, followed by the arithmetic logic unit (ALU) operation, and then the store operation. In a staggered scenario, the ALU operation of a first vector computational unit instruction may

execute in parallel with the load operation of the next vector computational unit instruction.

[0101] In various embodiments, different operations, including different arithmetic logic unit (ALU) operations, take one or more clock cycles to complete and there is no guarantee that the different operations complete by the end of the same clock cycle. In some embodiments, one or more of the fetch (801), decode (step 821), and issue (step 831) steps may be performed during the same instruction cycle.

[0102] At 841, a determination is made on whether the vector computational unit instruction includes a load operation. For example, in some scenarios, a load operation may be replaced with a no-op to indicate that no load operation should be performed. In response to a no-op, processing continues to 843. In the event that a load operation exists, processing continues to 845.

[0103] At 843, a no-op is processed and no load operation is performed. For example, a load instruction was not present in the instruction at 841 and instead the opcode for a no-op was used.

[0104] At 845, a load operation is executed by the vector computational unit. For example, a dequeue operation to load an input vector from a first-in-first-out queue, such as vector engine input queue 207, is performed.

[0105] At 851, a determination is made on whether the vector computational unit instruction includes an arithmetic logic unit (ALU) operation. For example, in some scenarios, an ALU operation may be replaced with a no-op to indicate that no ALU operation should be performed. In response to a no-op, processing continues to 853. In the event that an ALU operation exists, processing continues to 855.

[0106] At 853, a no-op is processed and no arithmetic logic unit (ALU) operation is performed. For example, an ALU instruction was not present in the instruction at 851 and instead the opcode for a no-op was used.

[0107] At 855, an arithmetic logic unit (ALU) operation is executed by the vector computational unit. For example, in response to a vector add operation, the arithmetic logic unit of a vector computational unit performs a vector add operation to add the contents of two source vector registers and store the result in a destination vector register. In some embodiments, the arithmetic logic unit of the vector computational unit is arithmetic logic units (ALUs) 427 of FIG. 4A.

[0108] At 861, a determination is made on whether the vector computational unit instruction includes a store operation. For example, in some scenarios, a store operation may be replaced with a no-op to indicate that no store operation should be performed. In response to a no-op, processing continues to 863. In the event that a store operation exists, processing continues to 865.

[0109] At 863, a no-op is processed and no store operation is performed. For example, a store instruction was not present in the instruction at 861 and instead the opcode for a no-op was used.

[0110] At 865, a store operation is executed by the vector computational unit. For example a store operation to store the vector data in a vector register to memory is performed.

[0111] FIG. 9 is a diagram illustrating an exemplary instruction cycle of a vector computational unit. The process of FIG. 9 illustrates an example ordering and sequence of three vector computational unit instructions performed in parallel but with staggered starts. In some embodiments, the exemplary instruction cycle of FIG. 9 is utilized by vector

engine 111, 211, 311, and/or vector computational unit 401 of FIGS. 1, 2, 3, and 4A, respectively. In the example of FIG. 9, the component instructions bundled as a single instruction are executed with staggered starts such that a load operation is executed first, followed by an arithmetic logic unit (ALU) operation, and then a store operation. In some embodiments, sequential vector computational unit instructions are pipelined but the component instructions are executed in parallel and do not follow the staggered starts depicted in FIG. 9.

[0112] In the example shown, a first instruction cycle 910 includes fetch step 911, a decode step 921, an issue step 931, a load execution step 941, an arithmetic logic unit (ALU) execution step 951, and a store execution step 961 corresponding to the first vector computational unit instruction. A second instruction cycle 920 includes fetch step 923, a decode step 933, an issue step 943, a load execution step 953, an arithmetic logic unit (ALU) execution step 963, and a store execution step 973 corresponding to the second vector computational unit instruction. A third instruction cycle 930 includes fetch step 935, a decode step 945, an issue step 955, a load execution step 965, an arithmetic logic unit (ALU) execution step 975, and a store execution step 985 corresponding to the third vector computational unit instruction. In some embodiments, the dotted vertical lines are clock cycle boundaries. In various embodiments, the steps within the same clock cycle boundaries are started during the same clock cycle.

[0113] In some embodiments, the start of instruction cycles are staggered by one stage. For example, first instruction cycle 910 is one stage ahead in processing compared to second instruction cycle 920, and two stages ahead of third instruction cycle 930. During any given clock cycle, different vector computational unit instructions can be utilizing the hardware resources associated with the different stages: fetch, decode, issue, load execution, arithmetic logic unit (ALU) execution, and store execution. As an example, issue stage 931, decode stage 933, and fetch stage 935 of first, second, and third instruction cycles 910, 920, and 930, respectively, execute during the same clock cycle. As another example, store execution step 961, arithmetic logic unit (ALU) execution step 963, and load execution step 965 of first, second, and third instruction cycles 910, 920, and 930, respectively, execute during the same clock cycle.

[0114] In some embodiments, the instruction cycle of a vector computational unit achieves a throughput of one vector computational unit instruction per clock cycle. In some embodiments, the fetch, decode, and/or issue steps are compressed into a single clock cycle. For example, in some embodiments, an instruction buffer is utilized to minimize fetch times and a fetch and decode step are performed together. In some embodiments, each stage of the instruction cycle may take one or more clock cycles to complete. In some embodiments, the stages are themselves pipelined. For example, in the event an execution step takes more than one cycle to complete, an execution step may be pipelined to complete over multiple clock cycles. In some embodiments, multiple execution steps may be processed in parallel in a pipelined manner and each execution step may correspond to a different vector computational unit instruction. In some embodiments, fetch steps 911, 923, and 935 correspond to step 801 of FIG. 8, decode steps 921, 933, and 945 correspond to step 821 of FIG. 8, issue steps 931, 943, and 955 correspond to step 831 of FIG. 8, load execution steps 941, 953 and 965 correspond to step 845 of FIG. 8, arithmetic

logic unit (ALU) execution steps **951**, **963**, and **975** correspond to step **855** of FIG. **8**, and store execution steps **961**, **973**, and **985** correspond to step **865** of FIG. **8**.

[**0115**] In an alternative embodiment (not shown), the fetch, decode, and issues stages of an instruction cycle are performed in the same order as FIG. **9**. In contrast with the exemplary embodiment of FIG. **9**, the load, arithmetic logic unit (ALU), and store execution steps are executed together and in parallel during the same clock cycle. For example, load execution step **941**, ALU execution step **951**, and store execution step **961** of the same vector computational unit instruction are executed together.

[**0116**] FIG. **10** is a block diagram illustrating an embodiment of a computation unit of a computational array. In the example shown, computation unit **1000** includes input values weight **1002**, data **1004**, and ResultIn **1006**; signals ClearAcc signal **1008**, Clock signal **1010**, ResultEnable signal **1012**, ResultCapture signal **1014**, and ShiftEn signal **1016**; components accumulator **1024**, multiplexer **1026**, shadow register **1028**, multiplier **1030**, and adder **1032**; logic **1034**, **1036**, and **1038**; and output value ResultOut **1050**. In some embodiments, logic **1034**, **1036**, and **1038** are AND gates. In some embodiments, additional signals are included as appropriate. In various embodiments, the computation unit of FIG. **10** is repeated for each of the plurality of computation units, such as computation unit **109**, of a computation array such as matrix processor **107** of FIG. **1**. Computation unit **1000** may be utilized to implement computational operations in parallel. In various embodiments, each computation unit of a computational array performs computations in parallel with the other computation units. In various embodiments, computation unit **1000** is a sub-circuit of a matrix processor that includes the functionality for performing one or more multiply, add, accumulate, and/or shift operations. For example, computation unit **1000** may be a sub-circuit that includes the functionality for performing a dot-product operation. In various embodiments, computation unit **1000** is computation unit **109** of FIG. **1** and/or computation units **209**, and/or **221-229** of FIG. **2**.

[**0117**] In some embodiments, Clock signal **1010** is a clock signal received by computation unit **1000**. In various embodiments, each computation unit of the computational array receives the same clock signal and the clock signal is utilized to synchronize the processing of each computation unit with the other computation units.

[**0118**] In the example shown, multiplier **1030** receives and performs a multiplication operation on the input values data **1004** and weight **1002**. The output of multiplier **1030** is fed to adder **1032**. Adder **1032** receives and performs an addition on the output of multiplier **1030** and the output of logic **1034**. The output of adder **1032** is fed to accumulator **1024**. In some embodiments, input values data **1004** and weight **1002** are lines that cross computation units and feed the corresponding data and/or weight to neighboring computation units. For example, in some embodiments, data **1004** is fed to all computation units in the same column and weight **1002** is fed to all computation units in the same row. In various embodiments, data **1004** and weight **1002** correspond to input elements fed to computation unit **1000** from a data input **103** and a weight input **105**, respectively. In various embodiments, data **1004** and weight **1002** correspond to input elements fed to computation unit **1000** from a data hardware data formatter and a weight hardware data formatter, respectively.

[**0119**] In some embodiments, ClearAcc signal **1008** clears the contents of accumulator **1024**. As an example, accumulation operations can be reset by clearing accumulator **1024** and used to accumulate the result of multiplier **1030**. In some embodiments, ClearAcc signal **1008** is used to clear accumulator **1024** for performing a new dot-product operation. For example, elements-wise multiplications are performed by multiplier **1030** and the partial-dot-product results are added using adder **1032** and accumulator **1024**.

[**0120**] In various embodiments, accumulator **1024** is an accumulator capable of accumulating the result of adder **1032** and indirectly the result of multiplier **1030**. For example, in some embodiments, accumulator **1024** is configured to accumulate the result of multiplier **1030** with the contents of accumulator **1024** based on the status of ClearAcc signal **1008**. As another example, based on the status of ClearAcc signal **1008**, the current result stored in accumulator **1024** may be ignored by adder **1032**. In the example shown, accumulator **1024** is a 32-bit wide accumulator. In various embodiments, accumulator **1024** may be sized differently, e.g., 8-bits, 16-bits, 64-bits, etc., as appropriate. In various embodiments, each accumulator of the plurality of computation units of a computational array is the same size. In various embodiments, accumulator **1024** may accumulate and save data, accumulate and clear data, or just clear data. In some embodiments, accumulator **1024** may be implemented as an accumulation register. In some embodiments, accumulator **1024** may include a set of arithmetic logic units (ALUs) that include registers.

[**0121**] In some embodiments, ResultEnable signal **1012** is activated in response to a determination that data **1004** is valid. For example, ResultEnable signal **1012** may be enabled to enable processing by a computation unit such as processing by multiplier **1030** and adder **1032** into accumulator **1024**.

[**0122**] In some embodiments, ResultCapture signal **1014** is utilized to determine the functionality of multiplexer **1026**. Multiplexer **1026** receives as input ResultIn **1006**, output of accumulator **1024**, and ResultCapture signal **1014**. In various embodiments, ResultCapture signal **1014** is used to enable either ResultIn **1006** or the output of accumulator **1024** to pass through as the output of multiplexer **1026**. In some embodiments, multiplexer **1026** is implemented as an output register. In some embodiments, ResultIn **1006** is connected to a computation unit in the same column as computation unit **1000**. For example, the output of a neighboring computation unit is fed in as an input value ResultIn **1006** to computation unit **1000**. In some embodiments, the input of a neighboring computation unit is the computation unit's corresponding ResultOut value.

[**0123**] In some embodiments, shadow register **1028** receives as input the output of multiplexer **1026**. In some embodiments, shadow register **1028** is configured to receive the output of accumulator **1024** via multiplexer **1026** depending on the value of ResultCapture signal **1014**. In the example shown, the output of shadow register **1028** is output value ResultOut **1050**. In various embodiments, once a result is inserted into shadow register **1028**, accumulator **1024** may be used to commence new calculations. For example, once the final dot-product result is stored in shadow register **1028**, accumulator **1024** may be cleared and used to accumulate and store the partial result and eventually the final result of a new dot-product operation on new weight and data input values. In the example shown, shadow

register **1028** receives a signal ShiftEn signal **1016**. In various embodiments, ShiftEn signal **1016** is used to enable or disable the storing of values in the shadow register **1028**. In some embodiments, ShiftEn signal **1016** is used to shift the value stored in shadow register **1028** to output value ResultOut **1050**. For example, when ShiftEn signal **1016** is enabled, the value stored in shadow register **1028** is shifted out of shadow register **1028** as output value ResultOut **1050**. In some embodiments, ResultOut **1050** is connected to a neighboring computation unit's input value ResultIn. In some embodiments, the last cell of a column of computation units is connected to the output of the computational array. In various embodiments, the output of the computational array feeds into a vector engine such as vector engine **111** of FIG. **1** for vector processing. For example, the output ResultOut **1050** of a computation cell such as computation cell **109** of FIG. **1** may be fed into a processing element of a vector engine such as processing element **113** of vector engine **111** of FIG. **1**.

[**10124**] In the example shown, shadow register **1028** is 32-bits wide. In various embodiments, shadow register **1028** may be sized differently, e.g., 8-bits, 16-bits, 64-bits, etc., as appropriate. In various embodiments, each shadow register of the plurality of computation units of a computational array is the same size. In various embodiments, shadow register **1028** is the same size as accumulator **1024**. In various embodiments, the size of multiplexer **1026** is based on the size of accumulator **1024** and/or shadow register **1028** (e.g., the same size or larger).

[**10125**] In some embodiments, logic **1034**, **1036**, and **1038** receive signals, such as control signals, to enable and/or configure the functionality of computation unit **1000**. In various embodiments, logic **1034**, **1036**, and **1038** are implemented using AND gates and/or functionality corresponding to an AND gate. For example, as described above, logic **1034** receives ClearAcc signal **1008** and an input value corresponding to the value stored in accumulator **1024**. Based on ClearAcc signal **1008**, the output of logic **1034** is determined and fed to adder **1032**. As another example, logic **1036** receives ResultEnable signal **1012** and Clock signal **1010**. Based on ResultEnable signal **1012**, the output of logic **1036** is determined and fed to accumulator **1024**. As another example, logic **1038** receives ShiftEn signal **1016** and Clock signal **1010**. Based on ShiftEn signal **1016**, the output of logic **1038** is determined and fed to shadow register **1028**.

[**10126**] In various embodiments, computation units may perform a multiplication, an addition operation, and a shift operation at the same time, i.e., within a single cycle, thereby doubling the total number of operations that occur each cycle. In some embodiments, results are moved from multiplexer **1026** to shadow register **1028** in a single clock cycle, i.e., without the need of intermediate execute and save operations. In various embodiments, the clock cycle is based on the signal received at Clock signal **1010**.

[**10127**] In various embodiments, input values weight **1002** and data **1004** are 8-bit values. In some embodiments, weight **1002** is a signed value and data **1004** is unsigned. In various embodiments, weight **1002** and data **1004** may be signed or unsigned, as appropriate. In some embodiments, ResultIn **1006** and ResultOut **1050** are 32-bit values. In various embodiments ResultIn **1006** and ResultOut **1050** are implemented using a larger number of bits than input operands weight **1002** and data **1004**. By utilizing a large

number of bits, the results of multiplying multiple pairs of weight **1002** and data **1004**, for example, to calculate a dot-product result, may be accumulated without overflowing the scalar result.

[**10128**] In some embodiments, computation unit **1000** generates an intermediate and/or final computation result in accumulator **1024**. The final computation result is then stored in shadow register **1028** via multiplexer **1026**. In some embodiments, multiplexer **1026** functions as an output register and store the output of accumulator **1024**. In various embodiments, the final computation result is the result of a convolution operation. For example, the final result at ResultOut **1050** is the result of convolution between a filter received by computation unit **1000** as input values using weight **1002** and a two-dimensional region of sensor data received by computation unit **1000** as input values using data **1004**.

[**10129**] As an example, a convolution operation may be performed using computation unit **1000** on a 2x2 data input matrix [d0 d1; d2 d3] corresponding to a region of sensor data and a filter corresponding to a 2x2 matrix of weights [w0 w1; w2 w3]. The 2x2 data input matrix has a first row [d0 d1] and a second row [d2 d3]. The filter matrix has a first row [w0 w1] and a second row [w2 w3]. In various embodiments, computation unit **1000** receives the data matrix via data **1004** as a one-dimensional input vector [d0 d1 d2 d3] one element per clock cycle and weight matrix via weight **1002** as a one-dimensional input vector [w0 w1 w2 w3] one element per clock cycle. Using computation unit **1000**, the dot product of the two input vectors is performed to produce a scalar result at ResultOut **1050**. For example, multiplier **1030** is used to multiply each corresponding element of the input weight and data vectors and the results are stored and added to previous results in accumulator **1024**. For example, the result of element d0 multiplied by element w0 (e.g., d0\*w0) is first stored in cleared accumulator **1024**. Next, element d1 is multiplied by element w1 and added using adder **1032** to the previous result stored in accumulator **1024** (e.g., d0\*w0) to compute the equivalent of d0\*w0+d1\*w1. Processing continues to the third pair of elements d2 and w2 to compute the equivalent of d0\*w0+d1\*w1+d2\*w2 at accumulator **1024**. The last pair of elements is multiplied and the final result of the dot product is now stored in accumulator **1024** (e.g., d0\*w0+d1\*w1+d2\*w2+d3\*w3). The dot-product result is then copied to shadow register **1028**. Once stored in shadow register **1028**, a new dot-product operation may be initiated, for example, using a different region of sensor data. Based on ShiftEn signal **1016**, the dot-product result stored in shadow register **1028** is shifted out of shadow register **1028** to ResultOut **1050**. In various embodiments, the weight and data matrices may be different dimensions than the example above. For example, larger dimensions may be used.

[**10130**] In some embodiments, a bias parameter is introduced and added to the dot-product result using accumulator **1024**. In some embodiments, the bias parameter is received as input at either weight **1002** or data **1004** along with a multiplication identity element as the other input value. The bias parameter is multiplied against the identity element to preserve the bias parameter and the multiplication result (e.g., the bias parameter) is added to the dot-product result using adder **1032**. The addition result, a dot-product result offset by a bias value, is stored in accumulator **1024** and later shifted out at ResultOut **1050** using shadow register **1028**. In

some embodiments, a bias is introduced using a vector engine such as vector engine **111** of FIG. **1**.

**[0131]** Although the foregoing embodiments have been described in some detail for purposes of clarity of understanding, the invention is not limited to the details provided. There are many alternative ways of implementing the invention. The disclosed embodiments are illustrative and not restrictive.

What is claimed is:

1. A microprocessor system, comprising:  
a computational array that includes a plurality of computation units; and  
a vector computational unit in communication with the computational array.
2. The system of claim **1**, wherein the vector computational unit includes a plurality of processing elements, and the processing elements are configured to receive output data elements from the computational array and process in parallel the received output data elements.
3. The system of claim **2**, wherein the processing elements process in parallel the received output data elements in response to a single processor instruction.
4. The system of claim **1**, wherein the computational array includes a matrix processor.
5. The system of claim **1**, wherein the computational array is configured to receive two vector input operands.
6. The system of claim **1**, wherein each computation unit of the plurality of computation units includes an arithmetic logic unit, an accumulator, and a shadow register.
7. The system of claim **1**, wherein each computation unit of the plurality of computation units is configured to perform a multiply operation and an add operation.
8. The system of claim **1**, wherein each computation unit of the plurality of computation units is configured to perform a dot-product component operation.
9. The system of claim **1**, wherein each computation unit of the plurality of computation units is configured to compute a dot-product result component in parallel in response to a single computational array instruction.
10. The system of claim **2**, wherein each processing element of the plurality of processing elements includes an arithmetic logic unit configured to perform arithmetic logic unit operations in parallel with other processing elements.
11. The system of claim **2**, wherein a notification signal identifies that output data elements from the computational array are ready for the vector computational unit.
12. The system of claim **1**, wherein the computational array is configured to operate as a first-in-first-out queue.
13. The system of claim **2**, wherein the output data elements from the computational array correspond to dot-product results.
14. The system of claim **2**, wherein the output data elements from the computational array correspond to convolution results performed on image data.
15. The system of claim **3** wherein the single processor instruction is used to calculate a result of a non-linear function.

**16.** The system of claim **15**, wherein the non-linear function is a rectified linear unit function or a sigmoid function.

**17.** The system of claim **1**, further comprising a post-processing unit in communication with the vector computational unit.

**18.** The system of claim **17**, wherein the post-processing unit is configured to perform a pooling function.

**19.** The system of claim **2**, wherein the received output data elements from the computational array are stored in an accumulator.

**20.** The system of claim **19**, wherein each processing element of the plurality of processing elements is configured to access a slice of the accumulator and a slice of one or more vector registers.

**21.** The system of claim **2**, wherein the vector computational unit further includes a plurality of vector registers sized to fit the output data elements from the computational array.

**22.** A microprocessor system, comprising:

a computational array that includes a plurality of computation units, wherein each computation unit of the plurality of computation units is configured to perform a dot-product component operation in response to a single computational array instruction; and

a vector computational unit in communication with the computational array, wherein the vector computational unit includes a plurality of processing elements and the processing elements are configured to receive output data elements from the computational array and process in parallel the received output data elements in response to a single vector computational unit instruction.

**23.** The system of claim **22**, further comprising:

a control unit configured to provide the single computational array instruction to the computational array and the single vector computational unit instruction to the vector computational unit.

**24.** The system of claim **23**, wherein the control unit synchronizes the output data elements transferred from the computational array to the processing elements of the vector computational unit.

**25.** The method comprising:

receiving a single processor instruction for a vector computational unit, wherein the vector computational unit is in communication with a computational array and includes a plurality of processing elements configured to receive output data elements from the computational array;

receiving the output data elements from the computational array, wherein the computational array includes a plurality of computation units; and

processing in parallel the received output data elements in response to the single processor instruction.

\* \* \* \* \*