

(51) International Patent Classification:
G06F 13/28 (2006.01)

(21) International Application Number:
PCT/US20 17/022975

(22) International Filing Date:
17 March 2017 (17.03.2017)

(25) Filing Language:
English

(26) Publication Language:
English

(30) Priority Data:
15/073,905 18 March 2016 (18.03.2016) US

(71) Applicant: ORACLE INTERNATIONAL CORPORATION [US/US]; 500 Oracle Parkway, Mail Stop 50P7, Redwood Shores, California 94065 (US).

(72) Inventors: BROWN, David A.; 500 Oracle Parkway, Mail Stop 50P7, Redwood Shores, California 94065 (US). JAIN, Rishabh; 500 Oracle Parkway, Mail Stop 50P7, Redwood Shores, California 94065 (US). DULLER, Michael; 500 Oracle Parkway, Mail Stop 50P7, Redwood Shores, California 94065 (US). IDICULA, Sam; 500 Oracle Parkway, Mail Stop 50P7, Redwood Shores, California 94065 (US). SCHLANGER, Erik; 500 Oracle Parkway, Mail Stop 50P7, Redwood Shores, California 94065 (US). HAWKINS, David Joseph; 500 Oracle Parkway, Mail Stop 50P7, Redwood Shores, California 94065 (US).

(74) Agents: BINGHAM, Marcel K. et al; Hickman Palermo Becker Bingham LLP, 1 Almaden Boulevard, San Jose, California 95113 (US).


Published: with international search report (Art. 21(3))

(54) Title: TUPLE ENCODING AWARE DIRECT MEMORY ACCESS ENGINE FOR SCRATCHPAD ENABLED MULTICORE PROCESSORS

(57) Abstract: Tabular data is moved efficiently from a source memory to a destination memory. A first set of electronic circuits accesses a descriptor stored at a particular memory location. The descriptor indicates a width of a column and a number of rows of tabular data, and a tabular data manipulation operation, and indicates a source memory location storing the tabular data and a destination memory location for storing the tabular data manipulation result. The first set of electronic circuits determines control information indicating the tabular data manipulation operations to be performed and transmits the control information to a second set of electronic circuits. Based on the control information, the second set of electronic circuits retrieve the tabular data from the source memory location and applies the tabular data manipulation operation to generate the data manipulation result, and causes the data manipulation result to be stored at the destination memory location.

FIG. 6

- In response to a particular memory location being pushed into a first register with a first register name that is accessible by a first set of electronic circuits
- Said first set of electronic circuits accesses a descriptor stored at the particular memory location
- The first set of electronic circuits determines, based on the descriptor, one or more instructions indicating one or more data manipulation operations to perform the tabular data
- The first set of electronic circuits manipulates, using a first function data stored, the one or more instructions to a second set of electronic circuits to perform the one or more operations
- According to the one or more instructions, said second set of electronic circuits retrieves said tabular data from a source memory location
- Apply said one or more data manipulation operations to said tabular data
- Cause said data manipulation result to be stored at said destination location.
BACKGROUND
[0001] Database servers that execute on multi-core processors perform data manipulation operations on large amounts of tabular data. Tabular data is data that is logically organized as rows and one or more columns, each column having a certain size, each row including each column. Logically, tabular data resides in a table-like structure, such as a spreadsheet or relational table. However, the actual physical storage of the tabular data may take a variety of forms. For example, in row-major format, tabular data may be stored as rows that are stored contiguously within a memory address space, each row including each column and a given column occupying the same number of bytes within a row. In column-major format, each column may be separately stored from other columns as a column vector stored contiguously within a memory address, the particular entry for each row in a column vector being stored in the same relative position or index of the respective column vector of each other column.

[0002] To perform data manipulation operations on tabular data efficiently, tabular data is moved from main memory to a memory closer to a core processor, where the operations can be performed more efficiently by the core processor. Thus, the movement of tabular data between the memory closer to a core processor and main memory is the type of operation that is performed frequently by database servers.

[0003] However, approaches for moving tabular data to a memory closer to the core processor add overhead that significantly offsets or eliminate any advantage gained by the movement of tabular data to the memory closer to the core processor. Even direct memory access (DMA) engines capable of offloading the task of moving data cannot offer sufficient increase in processor efficiency for several reasons. Tabular data processed by database operations is not organized or formatted in a way that is optimal for a DMA engine to move.
Additionally, the memory closer to the core processor is typically small in size. Therefore, a DMA engine will be able to move only a small portion of data into the local memory before that memory is full and needs to be emptied before it needs to be written to again. This results in the DMA engine repeating the process multiple times and issuing an interrupt each time the DMA moves data to the memory closer to the core processor, resulting in large number of interrupts. A large number of interrupts deteriorate core processor performance because every time the core processor is interrupted, the core processor must determine the source of the interrupt and how to handle the interrupt.

Furthermore, in multi-core processor architectures, where each core has its own local memory, a DMA engine is required per core in order to access the local memory of the core. Implementing a DMA engine per core dramatically increases the cost, in terms of gate count, area and power needed, of such multi-core processor architectures.

**BRIEF DESCRIPTION OF THE DRAWINGS**

In the drawings:

- Figure 1 illustrates an example arrangement of a plurality of electronic circuits of the data movement system.
- Figure 2 illustrates an example of descriptors to move data from a source memory location to a destination memory location.
- Figure 3 illustrates an example of descriptors for moving data stored in tabular format.
- Figure 4 illustrates an example of descriptors for performing data manipulation operations on one or more.
- Figure 5 illustrates an example arrangement of data manipulation operation blocks.
- Figure 6 illustrates an example process for moving data using hardware circuitry without execution of software.
- Figure 7A illustrates an example of columns that may be partitioned between core processors, according to an embodiment of the present invention.
- Figure 7B illustrates partitioning stages for partitioning data between core processors, according to an embodiment of the present invention.
- Figure 7C illustrates an example arrangement of a plurality of electronic circuits of parts of a data movement system that participate in partitioning, according to an embodiment of the present invention.
- Figure 7D illustrates an arrangement of descriptors for partitioning data, according to an embodiment of the present invention.
[0017] Figure 7E illustrates an example of columns that may be partitioned between core processors, and an example of columns generated in conjunction with partitioning, according to an embodiment of the present invention.

[0018] Figure 8A illustrates a buffer in scratchpad memory used for receiving rows of a column during partitioning, according to an embodiment of the present invention.

[0019] Figure 8B illustrates operations performed to buffer rows of a column received during partitioning, according to an embodiment of the present invention.

[0020] Figure 9A illustrates pipelining of descriptors, according to an embodiment of the present invention.

[0021] Figure 9B illustrates pipelined descriptors according to an embodiment of the present invention.

[0022] Figure 10 illustrates RID columns used for row resolution after partitioning columns according to an embodiment of the present invention.

[0023] Figure 11A illustrates descriptors used for generating RID columns used for row resolution after partitioning according to an embodiment of the present invention.

[0024] Figure 11B illustrates RID columns used for row resolution after partitioning columns according to an embodiment of the present invention.

[0025] Figure 12A depicts various memories used by data movement engines according to an embodiment of the present invention.

[0026] Figure 12B depicts RID memories used by data movement engines according to an embodiment of the present invention.

[0027] Figure 13 depicts a copy ring interconnected to various memories used by a data movement engine according to an embodiment of the present invention.

[0028] Figure 14 depicts a descriptor used to configure data movement between various memories coupled to a copy ring according to an embodiment of the present invention.

[0029] Figure 15 is a flow chart depicting operations for data movement along a copy ring between various memories coupled to the copy ring according to an embodiment of the present invention.

[0030] Figure 16 depicts a copy ring which is interconnected between various data movement engines and which is used to broadcast data among data movement engines.
DETAILED DESCRIPTION

[0031] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form in order to avoid unnecessarily obscuring the present invention.

GENERAL OVERVIEW

[0032] The techniques described herein provide for accessing tabular data at a source memory location and copying and storing the tabular data at a destination memory location without executing software programs. Additionally, the techniques also provide for performing one or more data manipulation operations, again without executing software programs, on the tabular data while the data is in flight to the destination memory location. In other words, data manipulation operations will typically be performed within the same clock cycles that are used in transmitting the data to the destination memory location and prior to storing the tabular data at the destination memory location. Consequently, the tabular data stored in the destination memory location is the tabular data resulting from the data manipulation operations. Therefore, the core processor can avoid spending additional clock cycles in manipulating the tabular data.

[0033] In an embodiment, in response to a particular memory location being pushed or stored into a first register within a first register space that is accessible by a first set of electronic circuits, the first set of electronic circuits access a descriptor stored at the particular memory location. The descriptor may indicate a source memory location of where the said tabular data is stored, and the descriptor may also indicate a destination memory location to store the result of a data manipulation operation. The destination memory location may be within a memory that is local to the core data processor that programmed the descriptor. With the memory being local to the core data processor, the distance between the core data processor and the memory is short, therefore the latency in accessing the tabular data from the destination memory location is reduced. Performance of the core processor in accessing the tabular data is improved.

[0034] The descriptor may also indicate a width of a column of tabular data, wherein the tabular data comprises columns and rows, a number of rows of tabular data. By describing the tabular data to be moved in terms of number of rows and width of the column of tabular data, the descriptor specifies to the data movement system how a database column of a database table is
formatted and stored at the source memory location. Different columns of the database table may be specified by different descriptors, thus the data movement system is fully aware of how database tables are formatted and stored at the source memory location. Therefore, the data movement system is always optimized to access tabular data from a source memory location and store it in a destination memory location, depending on how the tabular data is formatted and organized at the source memory location.

[0035] The descriptor may also indicate one or more tabular data manipulation operations to perform on the column of data. An example of a tabular data manipulation operation may be a type of filtering operation, described herein as a gather operation. The descriptor may indicate that a gather operation should be performed on the tabular data. The descriptor may also specify a memory location of a bit vector within the local memory. The data movement system, described herein, uses the bit vector specified by the descriptor to filter out rows of the column of tabular data that do not satisfy the filtering criteria. The data movement system filters out rows of the column of tabular data while the data is in flight to the destination memory location or prior to the tabular data being stored in the destination memory location, and compacts the resulting rows, the rows that were not filtered out, while storing the resulting rows in the destination memory location such that the resulting rows are stored in consecutive memory locations within destination memory even if the resulting rows were not stored in consecutive memory locations at the source memory location.

[0036] Therefore, the number of rows of the column of tabular data stored in the destination memory location after a gather operation is performed will likely be fewer than the number of rows specified by the descriptor. Thus, in addition to performance gain mentioned above, a more efficient usage of the limited storage space within a memory local to a core in multi-core processor is achieved.

[0037] Based on the descriptor, the first set of electronic circuits determine control information that indicate one or more data manipulation operations to perform on the tabular data. Using a hardware data channel, the first set of electronic circuits transmit the control information to a second set of electronic circuits to perform the one or more operations. The second set of electronic circuits, according to the control information, retrieve the tabular data from a source memory location, and apply the one or more data manipulation operations to the
tabular data to generate the data manipulation result and cause the data manipulation result to be stored at the destination memory location.

[0038] Once the tabular data is stored at the destination memory location, the techniques described herein allow the data movement system to notify the core processor that data is available in the destination memory by using event identifiers that are associated with a particular descriptor. The core processor is capable of handling these event identifiers without needing interrupt handlers. Thus, significantly reducing overhead costs of interrupt handlers.

[0039] The techniques described herein further allow the data movement system to begin processing the next descriptor. The data movement system may begin processing the next descriptor while the core processor processes the tabular data stored at the destination memory location. In other words, processing of the next descriptor and accessing and processing of tabular data stored may occur simultaneously. Therefore, the core processor may access and process data from the destination memory while the data movement system is moving tabular data for the next descriptor to a different destination memory location specified in that descriptor. Thereby significantly improving the speed of processing data.

[0040] In other embodiments, the disclosure encompasses a data movement system and a computer apparatus to carry out the foregoing steps.

OVERVIEW OF THE DATA MOVEMENT SYSTEM

CORE PROCESSOR AND DMEM

[0041] FIG. 1 illustrates an example arrangement of a Data Movement System 101. FIG. 1 comprises a plurality of core processors 103a, 103g, 104a, 104g. Each of the core processors 103a, 103g, 104a, 104g are connected to a local high-speed scratchpad memory, such as a static random-access memory (SRAM), referred to herein as DMEM (Direct Memory). In FIG. 1, core processors 103a, 103g, 104a, 104g are connected to DMEM units 102a, 102g, 105a, 105g, respectively. Of all the core processors, only the processor that is connected to a particular DMEM may directly access that particular DMEM. Thus, DMEM 102a may be accessed by core processor 103a, but cannot be accessed by core processor 103g, 104a, 104g. DMEM 102g may be accessed by core processor 103g, but not by core processors 103a, 104a, 104g. Likewise, DMEM 105a may be accessed by core processor 104a, but not by core processors 103a, 103g, 104g and DMEM 105g may be accessed by core processor 104g, but cannot be accessed by core processors 103a, 103g, 104a.
DIRECT MEMORY ACCESS COMPLEX (DMAC)

[0042] The data movement system described herein comprises three major functional blocks, Direct Memory Access Complex (DMAC), Direct Memory Access X-Bar (DMAX) and Direct Memory Access DMEM (DMAD). The data movement system described herein comprises only one DMAC block and several DMAX and DMAD blocks. The DMAC comprises several complex data retrieval, load and manipulation engines. The DMAX blocks mainly comprise data routing engines and the DMAD blocks mainly comprise data decoding engines and descriptor channel blocks.

[0043] The data movement system described herein comprises one DMAD block per core, therefore the number of the DMAD blocks depend upon the number of cores utilized in the data movement system. For example, a data movement system in a 32 core processor system, the number of DMAD blocks is 32. In an embodiment, several DMAD blocks may be configured to route data and instructions to one DMAX block. Continuing with the example of the 32 core processor, 8 DMAD blocks may be configured to route data and instructions to one DMAX block, thereby resulting in 4 DMAX blocks to serve 32 DMAD blocks of 32 cores.

[0044] The data movement system described herein comprises only a single DMAC. The single DMAC processes data and instructions, routed via DMAX, from all DMAD blocks of the data movement system. The DMAC comprises engines that perform complex functions and due to their complexity, require the highest gate count, area and power relative to the engines within DMAX and DMAD blocks. Therefore, the DMAC impacts the total cost of the data movement system more than DMAX and DMAD blocks. However, by sharing the DMAC resources across the DMAX and DMAD blocks, the cost of a DMAC to the data movement system is amortized. Thus, the total cost of the data movement system, in terms of gate count, area and power, is substantially lower than alternative approaches described above.

[0045] In an embodiment, the data movement system described herein is implemented on a single chip. Thus, for each core processor, the DMEM connected to each of the core processors, the DMAD block for each of the core processors, DMAX blocks and the DMAC block are all designed, implemented and configured on a single chip. A bus interconnects all the functional blocks of the data movement system in the chip.
FIG. 1 depicts Direct Memory Access Complex (DMAC) 140 and several major engines of DMAC 140. The details and workings of DMAC 140 and its engines are described further below.

DIRECT MEMORY ACCESS DMEM (DMAD)

Each of the core processors, 103a, 103g, 104a, 104g, is connected to DMAD 106a, 106g, 115a, 115g, respectively. Each DMAD comprises a group of electronic circuits that have been designed to receive instructions from the core processor connected to the particular DMAD. For example, DMAD 106a is designed to receive instructions from core processor 103a only.

A core processor sends instructions to a DMAD by programming a set of commands, herein referred to as a descriptor. A descriptor describes movement of data from one location to another location through a plurality of fields. Some of the fields in the descriptor may include a descriptor type, a source address location that indicates the source location for the tabular data to be moved from, a destination address location that indicates the destination location for the tabular data from the source location to be copied to, the size of the column of tabular data to be operated on, the number of rows of the column of tabular data that need to be copied, one or more data manipulation operations and wait-for event identifiers and other control flags.

Once the core processor programs the descriptor, the core processor stores the descriptor at a location in the DMEM. For example, core processor 103a upon programming the descriptor, stores it in DMEM unit 102a. Core processor 103a then sends the descriptor to DMAD 106a by transmitting the memory location of the descriptor within the DMEM unit 102a onto one of the two hardware data channels of DMAD 106a. A core processor transmits the memory location of a descriptor onto a data channel by storing the memory location into a register. In an embodiment, the register may be designed to be a first-in-first-out or FIFO register such that the first memory location that is pushed or stored into the register will be the first memory location that is added into a hardware managed list of one of the two hardware data channels.

DMEM INTERFACE BLOCK

Each DMAD comprises a DMEM interface block that is configured to store any data destined for the DMEM unit coupled with its DMAD, and generate a write request for the DMEM unit coupled with its DMAD to store the data transmitted to its DMAD in the DMEM unit coupled with its DMAD. For example, DMAD 106a comprises DMEM interface block
107a. DMEM interface block 107a is a group of electronic circuits that have been designed to store data transmitted to DMAD 106a and destined for DMEM unit 102a in one of the registers accessible by DMEM interface block 107a. Additionally, the group of electronic circuits of DMEM interface block 107a have also been designed to generate a write request, for DMEM unit 102a, to store the data destined for DMEM unit 102a. DMAD 106g, 115a, and 115g similarly comprise DMEM interface blocks 107g, 109a, 109g respectively.

[0051] The DMEM interface block is also configured to read or retrieve data from the DMEM unit coupled with its DMAD. The DMEM interface block may generate a read request, using a DMEM unit location, to read or retrieve data from the DMEM unit location. The DMEM interface block may receive a read request to read or retrieve data from a particular DMEM unit location and in response the DMEM interface block may read or retrieve data from the particular DMEM unit location. The DMEM interface block may transmit the read or retrieved data to the hardware component within its DMAD that requested that data. The DMEM interface block may receive a write request to write or store data at a particular DMEM unit location and in response the DMEM interface block may write or store data at the particular DMEM location in the DMEM unit coupled with the DMAD of the DMEM interface block. Each of DMEM interface blocks 107a, 107g, 109a, 109g, depicted in FIG. 1, are designed to perform the above operations with DMEM units 102a, 102g, 105a, and 105g, respectively.

DESCRIPTION CHANNEL BLOCK OF DMAD

[0052] Each DMAD comprises a Descriptor Channel Block, which is a subset of electronic circuits of the DMAD that are designed to determine the hardware data channel to which the descriptor will be added. In an embodiment, each DMAD may be designed to maintain two hardware data channels, and may have two Descriptor Channel Blocks, one for each of the hardware data channels. For example, DMAD 106a is designed to maintain two hardware data channels. Descriptor Channel Blocks 108a and 108b are the two descriptor channel blocks of DMAD 106a. Similarly, DMAD 106g comprises Descriptor Channel Blocks 108g, 108h, DMAD 115a comprises Descriptor Channel Blocks 111a, 111b and DMAD 115g comprises Descriptor Channel Blocks 111g, 111h.

[0053] Each Descriptor Channel Block maintains two hardware managed lists, an active list and a free list, per hardware data channel. In an embodiment, the hardware managed active list and free list are linked lists. Once the core processor stores the DMEM location of the descriptor
into the FIFO register, the Descriptor Channel Block of the DMAD connected to the core processor transmits the DMEM location of the descriptor from the FIFO register to one of the hardware data channels. In an embodiment, once the DMEM location of a descriptor is stored into a FIFO register, the Descriptor Channel Block determines the number of descriptors that are assigned to be processed on that particular hardware data channel and if the number of descriptors that are assigned to be processed on that particular hardware data channel is greater than zero, then the Descriptor Channel Block adds the new descriptor identified by the newly pushed or stored DMEM location in the FIFO register to the active list of that particular hardware data channel. The Descriptor Channel Block adds the new descriptor to the active list by transmitting instructions to the DMAD to write the DMEM location of the new descriptor to the Link Address field of the last descriptor that was added to that particular hardware data channel.

[0054] The Descriptor Channel Block begins processing a descriptor by storing the DMEM location of the descriptor into a register that has been designated as the register from which the Descriptor Channel Block is designed to read from and start processing the next available descriptor, referred herein as the Next Descriptor to Read register. If the active list is empty, then the Descriptor Channel Block stores the DMEM location from the FIFO register into the Next Descriptor to Read register. If the active list is not empty, then the Descriptor Channel Block adds the descriptor, stored at the DMEM location from the FIFO register, to the end of the active list by updating the Link Address field value of the descriptor previously at the end of the active list to contain the DMEM location value from the FIFO register.

[0055] In an embodiment, a register, described herein as the Last Descriptor List register, accessible by the Descriptor Channel Block comprises the DMEM location of the descriptor that is currently at the end of the active list. The Descriptor Channel Block adds a new descriptor to the active list by storing or writing the DMEM location from the FIFO register as the value of the Link Address field of the descriptor currently at the end of the list and storing the DMEM location value from the FIFO register in the Last Descriptor List register. The Descriptor Channel Block then traverses through the active list using the Link Address field of the descriptor that is currently being processed.

[0056] Once the DMEM location of a Descriptor is stored in the Next Descriptor to Read register, the Descriptor Channel Block, using the DMEM location stored in the register, retrieves
the data of the descriptor available at that DMEM location from the DMEM. The Descriptor
Channel Block transmits a request to read data from DMEM to the DMEM Interface Block of
the DMAD. The request to read data includes the DMEM location of the descriptor. In an
embodiment, the request to read data also specifies a number of bytes to read. In an embodiment
the number of bytes to read equals the number of bytes that make up the entire descriptor or the
total size of the descriptor. In an embodiment, the total size of a descriptor is 16 bytes. The
DMEM Interface Block retrieves data from DMEM using the specified DMEM location and
forwards the data to the Descriptor Channel Block. The Descriptor Channel Block decodes the
descriptor data including, but not limited to, determining the type of the descriptor. The
Descriptor Channel Block determines the type of the descriptor and processes the descriptor
based at least on the type of the descriptor.

DESCRIPTOR TYPES

[0057] According to an embodiment, there are at least three types of descriptors, which are
data descriptors, control descriptors, auxiliary descriptors. The type of the descriptor is indicated
by a descriptor type field within the descriptor data. There are multiple variations within each
type of descriptor. Data descriptors specify how Data Movement System moves data from one
memory location to another memory location, and the data is transformed during movement.

[0058] Control descriptors provide information for looping through one or more descriptors
more than once. Additional control descriptors include (1) descriptors that may be used to
program certain configurations within the data movement system, referred to herein as program
descriptors, (2) descriptors that may be used to control event registers in the data movement,
referred to herein as event descriptors, and (3) descriptors that may assist with partitioning of
tabular data, referred to herein as hash and range engine descriptors.

[0059] Auxiliary descriptors provide information that assist in the processing of another
descriptor. For example, the auxiliary descriptor may be used to provide additional control
information if the size of the required control information exceeds more than the maximum size
allowed for the control information.

[0060] Data descriptors, auxiliary descriptors and control descriptors that affect registers or
control state in the Direct Memory Access Complex (DMAC) 140 are forwarded to DMAC.
Control descriptors that affect the registers in a DMAD, that indicate loop mechanisms of one or
more descriptors, or other control descriptors that do not need to be sent to DMAC, or that affect
registers designed to store data corresponding to wait for events are further processed by the
Descriptor Channel Block.

DIRECT MEMORY ACCESS X-BAR (CROSS-BAR)

Descriptors are forwarded to DMAC by forwarding the data of the descriptors to
Direct Memory Access Cross(X)-Bar (DMAX) 110a, 110d. DMAX comprises electronic circuits
that are configured to control and route data flow from a DMAD to a DMAC and from the
DMAC to the DMAD. In an embodiment, the electronic circuits of a DMAX may be grouped
into 3 groups. One group of electronic circuits may be designed to transmit all control
information of descriptors from the DMAD to the DMAC, while another group of electronic
circuits may be designed to transmit, from the DMAD to the DMAC all data corresponding to a
response of a read request from the DMAC to the DMAD to read data from the DMEM. The
third group of electronic circuits may be designed to transmit a read request from DMAC to
DMAD to read data from the DMEM. Additionally, the third group of electronic circuits may be
designed to transmit all descriptors return paths from the DMAC to the DMAD, wherein each
descriptor return path comprises identifiers associated with a descriptor that indicate the DMAD
to which the descriptor belongs to, the descriptor channel block within that DMAD that
processed the descriptor and an identifier of that descriptor.

For example, DMAX 110a comprises an arbitration unit, such as the arbitration unit
112a and a FIFO register 112b for transmitting data from DMAD 106a to DMAC 140. In an
embodiment, data includes control information of a descriptor which may be used by the
arbitration unit 112a in selecting one of the input data paths and transmitting data including the
control information into the FIFO register 112b. Similarly, DMAX 110a comprises FIFO register
114b and routing unit 114a to transmit data from the DMAC to the DMAD. In an embodiment,
data transmitted from the DMAC may comprise control information such that routing unit 114a
selects the data path for the target DMAD to transmit the data. DMAX 110a also comprises
another arbitration unit 113a and a FIFO register 113b for transmitting data to be copied from
DMEM to an external storage memory.

DMAX 110d comprises arbitration units 112g and 113g and routing unit 114g that
provide the same functionality and perform the same functions as arbitration units 112a and 113a
and routing unit 114a, respectively. DMAX 110d also comprises FIFO registers 112h, 113h and
114h that provide the same functionality and perform the same functions as 112b, 113b, 114b respectively.

DIRECT MEMORY ACCESS COMPLEX (DMAC) - WRITE DESCRIPTOR PARSER

[0064] DMAC 140 comprises a write descriptor arbitration unit 120a, and the output of the write descriptor arbitration unit 120a is stored in the write descriptor parser logic block 120b. Write descriptor parser logic block 120b comprises one or more registers. Electronic circuits of write descriptor parser logic block 120b are designed to accumulate descriptor data and control information transmitted from a DMAX. In an embodiment, descriptor data from the DMAX may be transmitted in fragments, and electronic circuits of write descriptor parser logic block 120b may accumulate the various descriptor fields and reassemble the descriptor fields to form the complete descriptor data. Write descriptor parser logic block 120b determines the descriptor type of the descriptor and performs operations based on the descriptor type and the control information provided by the originating DMAD.

[0065] In response to determining that the descriptor is a data descriptor and in particular a write descriptor, write descriptor parser logic block 120b may modify the source address specified in the descriptor data using a source counter value provided by the originating DMAD. Additionally, write descriptor parser logic block 120b may also modify the destination address using a destination counter value provided by the originating DMAD. Write descriptor parser logic block 120b also transmits a data movement operation and the descriptor data to an appropriate data movement engine such as a DMEM load engine.

[0066] If the descriptor type is an auxiliary type descriptor, then write descriptor parser logic block 120b may update a local auxiliary data holding register and return the descriptor back to the originating DMAD. If the descriptor type is a program or control type descriptor, then write descriptor parser logic block 120b may store DMAC configuration data specified within the descriptor in the DMAC configuration register specified in the descriptor, and return the descriptor back to the originating DMAD.

DIRECT MEMORY ACCESS COMPLEX (DMAC)- READ DESCRIPTOR PARSER

[0067] DMAC 140 also comprises a read descriptor arbitration unit 121a, and the output of the read descriptor arbitration unit 121a is read descriptor parser logic block 121b. Read descriptor parser logic block 121b comprises one or more registers. Electronic circuits of read descriptor parser logic block 121b are designed to accumulate descriptor data and control
information transmitted from a DMAX. In an embodiment, descriptor data from a DMAX may be transmitted in fragments, and electronic circuits of read descriptor parser logic block 121b may accumulate the various descriptor fields and reassemble the descriptor fields to form the complete descriptor data. Read descriptor parser logic block 121b determines the descriptor type of the descriptor and performs operations based on the descriptor type and the control information provided by the origination DMAD.

[0068] In response to determining that the descriptor is a data descriptor and in particular a read descriptor, read descriptor parser logic block 121b may modify the source address specified in the descriptor data using a source counter value provided by the originating DMAD. Additionally, read descriptor parser logic block 121b may also modify the destination address using a destination counter value provided by the originating DMAD. Read descriptor parser logic block 121b also transmits a data movement operation and the descriptor data to an appropriate data movement engine such as a DDR load engine.

[0069] Similar to write descriptor parser logic block 120b, if the descriptor type is an auxiliary type descriptor, then read descriptor parser logic block 121b may update a local auxiliary data holding register and return the descriptor back to the originating DMAD. If the descriptor type is a program or control type descriptor, then read descriptor parser logic block 121b may store DMAC configuration data specified within the descriptor in the DMAC configuration register specified in the descriptor, and return the descriptor back to the originating DMAD.

DIRECT MEMORY ACCESS COMPLEX (DMAC) - DATA MOVEMENT ENGINES

[0070] DMAC 140 comprises data movement engines 130a, 130b, 130c, 130d. Each of the data movement engines 130a, 130b, 130c, 130d, comprise one or more DMEM load engines and one or more DDR load engines. Each of the data movement engines 130a, 130b, 130c, 130d also comprise one or more DMEM store engine and one or more DDR store engines. Each data movement engine receives operations from write descriptor parser logic block 120b, and read descriptor parser logic block 121b. Data movement engines 130a, 130b, 130c, 130d execute these operations by copying data from the specified source memory and storing data in the specified destination memory. Each data movement engine also uses control information provided by the descriptor parser logic block to their execute operations.
[0071] Data movement engines 130a, 130b, 130c, 130d generate read requests to the specified source memory. Data movement engines 130a, 130b, 130c, 130d accumulate data transmitted to the data movement engine in response to the read request, and then generate write requests to the specified destination memory. In an embodiment, a buffering process is implemented such that data transmitted to data movement engines may be stored in a register block accessible by the data movement engines. Data movement engines begin processing data transmitted in response to the read request without waiting for the requested data to be available.

[0072] Electronic circuits of system bus interface master block 123 are designed to receive read and write requests from the data movement engines 130a, 130b, 130c, 130d and translate them into system bus interface read requests and system bus interface write requests for memory units external to the data movement system, such as main memory or another memory unit. Electronic circuits of system bus interface master block 123 transmits data it receives in response to system bus interface read requests to the data movement engine that transmitted the read request. In an embodiment the system bus interface is AXI (Advanced Extensible Interface) and system bus interface master block 123 is an AXI master block.

[0073] Descriptor return block 125 is designed to return descriptors processed by write descriptor parser logic block 120b, read descriptor parser logic block 121b, and data movement engines 130a, 130b, 130c, 130d, to their originating DMAD.

[0074] DMS memory 150 comprises memory that various components of DMAC 140 may read from or write to. In general, DMS memory 150 is used to store data used by or generated by operations performed by the DMAC 140.

[0075] Supplemental Operation Engines 126 is representative of blocks of logic, each block performing a specific kind of operation on columns stored in DMS memory 150. For example, Supplemental Operation Engines 126 may include a partitioning engine that partitions tuples stored in one or more columns stored in DMS memory 150 among core processors in the data movement system. Such partitioning may include generating for each tuple an identifier identifying a co-processor to which the tuple is assigned by partitioning.

[0076] In addition, Supplemental Operation Engines 126 may include a hash engine. The hash engine generates hash values for one or more columns stored in DMS memory 150. Another example of an engine that may be included is a copy engine. The copy engine copies data between memory locations within DMS memory 150.
MOVING DATA AND PERFORMING DATA MANIPULATION OPERATIONS USING DESCRIPTORS

[0077] FIG. 2 illustrates an example method of moving data from a source memory location to a destination memory location using descriptors. FIG. 2 comprises three descriptors, 201a, 202a, 203a. Elements 201b, 202b and 203b each correspond to operations performed for descriptors 201a, 202a, 203a, respectively. The purposes of these elements is depict the order of operations performed for descriptors 201a, 202a, 203a.

[0078] FIG. 2 depicts a movement of ten thousand rows of data from a source memory location to a target memory location. In this example the source memory location is a double data rate synchronous dynamic random-access memory (DDR) and the target memory location is the DMEM connected to the core processor that programmed the descriptor, 102a and 103a respectively. In the example depicted in FIG. 2, descriptors 201a, 202a, 203a are programmed by core processor 103a.

DECODING DESCRIPTOR DATA

[0079] Descriptors 201a and 202a are data descriptors. The descriptor type field of the descriptors indicates that descriptors 201a and 202a are data descriptors. In an embodiment, binary numbers may be used to depict each descriptor type and direction in which the data is to be moved. For example, binary number 0000 may be encoded in the electronic circuits of the descriptor channel block of the DMAD that is processing the descriptors to represent data movement from DDR memory to DMEM memory or DMEM. Similarly, data movement from DMEM to DDR memory may be represented by binary number 0001. For descriptors 201a and 202a, data is to be moved from DDR memory to DMS memory or DMEM memory. Therefore, descriptor type field of descriptor 201a, 202a indicate the appropriate field value. The value of the “Desc Type” field shown in figure 201a and 202a is only for providing a clear illustrative example.

[0080] The core processor determines the source location of the source data along with the destination location of where the data is to be transmitted. The core processor also determines the number of rows that are to be processed at the source data location by a descriptor. In an embodiment, the core processor may be configured with a maximum number of rows that a descriptor is allowed to process. Such threshold limitation may be dynamically determined based on the size of DMEM or the available storage space in DMEM.
In FIG. 2, since the total number of rows of data that are to be processed is at least ten thousand rows, the core processor also programs a control descriptor that allows a DMAD to utilize the same descriptor numerous times. In other words, the control descriptor allows electronic circuits of the DMAD to implement a loop mechanism until some condition within the control descriptor is not satisfied. Control descriptors that allow the DMAD to implement such a loop mechanism will be referred to herein as loop descriptors.

In an embodiment, a core processor may also be configured to utilize multiple buffers in the DMEM to store data from the source data location. Utilization of multiple buffers allows for the core processor to access the data stored in the DMEM faster and consequently process that data faster than using a single buffer because it allows the core processor to access data stored in one buffer while the data movement system is moving or storing data in the other buffers. The flexibility of specifying different destination memory locations in different descriptors allows for the utilization of multiple buffers.

As described herein, a buffer is said to be associated with a descriptor if the destination memory location specified in the descriptor is the starting memory location of the buffer. Each descriptor may represent only a fraction of the total number of rows of a column of tabular data that is being moved into a DMEM unit. Thus, the buffer associated with a particular descriptor stores the fraction of the total number of rows of the column of tabular data and the core processor may begin processing the rows stored in the buffer without waiting for remaining number of rows of the column of tabular data being moved or stored into their respective buffers.

Additionally, the overhead costs from interrupt routines and interrupt handlers in switching control between the hardware components of the data movement system and the software executing on the core processor may be reduced by utilizing wait-for-events. The core processor may be configured to assign a particular event to a particular buffer in the DMEM and the values of the particular event will determine whether the hardware components of the data movement system will have access to the particular buffer or whether the software executing on the core processor will have access to the particular buffer.

In FIG. 2, descriptor 201a is assigned EventO. Based on the configuration, core processor 103a may either set EventO to a value of 1 or 0 in order to allow the hardware components of the DMAD to process the descriptor. For example, if the electronic circuits of the DMAD 106a have been designed to begin the processing of the descriptor only if EventO is set to
be zero, then core processor 103a will set the EventO value to 0 after core processor 103a programs the descriptor. Core processor 103a does not access that particular buffer until the value of EventO is set to one. DMAD 106a will set the value of EventO to 1 when the BufferO 201b is full.

[0086] In FIG. 2, within the DMEM unit 102a, the data is being stored in two buffers, one at address 0x0000 and another at address 0x2000. As described above, using at least two buffers enables faster processing of data. Once the DMAD 106a and other hardware components of data movement system begin processing descriptor 201a, data associated with that descriptor will be stored in BufferO at address 0x0000 of DMEM unit02a. Once BufferO has been filled with data, DMAD 106a will set the EventO value to 1, which will indicate to core processor 103a that BufferO is ready to be accessed and data in BufferO is ready to be processed. After the processing of descriptor 201a is completed the first time, DMAD 106a and other hardware components of the data movement system will begin processing descriptor 202a. While the hardware components of the data movement system begin processing descriptor 202a, core processor 103a will be processing data from BufferO. Therefore, using two data buffers allows for processing data records on a subset of data records without waiting for the entire set of data records to be retrieved first. Thus, reducing processing time and increasing processing speed.

[0087] In FIG. 2, descriptor 201a will be the first descriptor to be pushed on to one of the two hardware data channels of DMAD 106a and it will be the first descriptor among descriptors 201a, 202a, 203a to be processed. Descriptor 202a will be processed after descriptor 201a has begun processing and then descriptor 203a will be the last descriptor among the three descriptors to be processed, and descriptor 203a will be processed after descriptor 202a has begun processing. The core processor stores a descriptor in DMEM after programming the descriptor and in FIG. 2 core processor 103a stores descriptor 201a at address 0x5000, descriptor 202a at address 0x5010 and descriptor 203a at address 0x5020 of DMEM or DMEM unit 102a.

[0088] In FIG. 2, the "Src Addr" of descriptors 201a and 202a indicates the starting location of the column of data within the source memory where the tabular data is stored. "Dest Addr" of descriptors 201a and 202a indicates the location in DMEM where the data from the source memory will be stored. "Column Width" indicates the size of the data in the column of data in bytes and "Rows" indicates the number of rows that will be processed each time the data movement system is processing the descriptor 201a or 202a. A descriptor may comprise a "Src
Auto Inc Allow" field, wherein the "Src Auto Inc Allow" field indicates to a descriptor parser logic block within the DMAC to modify the source address based on values of one or more other fields within the descriptor. In an embodiment, the one or more other fields within the descriptor include, the "Counter Inc" field, the "Column Width" field and the "Rows" field of the descriptor. The descriptor parser logic block may modify the source address specified in the descriptor using the source address specified in the descriptor as a starting point or a base source address and adding an offset value to the base source address, wherein the offset value is determined by the descriptor parser logic block based on the values of the source counter, the width of the column of tabular data that is being moved or copied from the source address specified in the descriptor and the number of rows of the column of tabular data that is being moved or copied from the source address specified in the descriptor.

[0089] As described above, the "Column Width" field of the descriptor specifies the width of the column of the tabular data and the "Rows" field of the descriptor specifies the number of rows of the column of tabular data. The value of the source counter may be read or retrieved from a register comprising the source counter. In some embodiments, the "Counter Inc" field of a descriptor specifies the register that comprises the source counter value. In some embodiments, the "Counter Inc" field indicates that the counter value that should be considered or used is the source counter value and the descriptor channel block is configured to retrieve the value stored in a particular register that comprises the source counter value. In some embodiments, the descriptor channel block is designed to retrieve a source counter value from a particular register that has been designated to store source counter value.

[0090] The value of the "Src Addr Inc" field determines whether or not a counter specified by the "Counter Inc" field should be incremented. In an embodiment, if the "Src Addr Inc" field is set then the counter specified by the "Counter Inc" field is incremented by a descriptor channel block processing the descriptor, and if the "Src Addr Inc" field is not set then the counter specified by the "Counter Inc" field is not incremented. In an embodiment, the descriptor channel block that is processing the descriptor increments the value of the counter specified by the "Counter Inc" field of a source counter associated with a descriptor channel block by the descriptor channel block.

[0091] In FIG. 2, the "Counter Inc" field of descriptor 201a specifies that the counter is the source counter of the descriptor channel block processing descriptor 201a, which in FIG. 2, as
described above, is descriptor channel block 108a. The "Src Addr Inc" field of descriptor 201a triggers the incrementing of the counter value specified by the "Counter Inc" field by the descriptor channel block 108a. The descriptor channel block 108a increments the value of the counter specified by the "Counter Inc" field after the descriptor channel block has transmitted the control information of descriptor 201a to the DMAX associated with the DMAD of the descriptor channel block 108a.

"Link Addr" field helps the electronic circuits within a descriptor channel block of a DMAD maintain a variety of data structure in hardware. "Link Addr" field identifies the DMEM location where the next descriptor that the descriptor channel block must process is stored. Additionally the "Link Addr" field is not programmed by the software, instead the electronic circuits of the descriptor channel block, as described below, will determine the memory location of the next descriptor that should be processed and store the value of that memory location in the "Link Addr" field.

Loop address field of a control descriptor, specifically a loop descriptor, allows the electronic circuits within a descriptor channel block of a DMAD to implement the loop mechanism. In FIG. 2 the "Loop Addr" field of descriptor 203a contains the memory address value of descriptor 201a, thereby causing the descriptor channel block of the DMAD to reprocess descriptor 201a. After reprocessing descriptor 201a, the descriptor channel block of the DMAD will process the descriptor of the memory address stored in the "Link Addr" field of descriptor 201a, which means the descriptor channel block will reprocess descriptor 202a. Thus, the descriptor channel block will continue to reprocess all the descriptors until the loop condition is satisfied. The loop condition in descriptor 203a is specified by "Iteration Count". In FIG. 2 the loop condition is satisfied when the iteration count equals zero.

A descriptor channel block stores the iteration count specified in a loop descriptor in a particular register designated for storing iteration count values. Each time the descriptor channel block processes the loop descriptor, the descriptor channel block reads or retrieves the iteration count value from the particular register and determines whether it is zero or not. If the iteration count value is not zero, then the descriptor channel block processes the loop descriptor, decrements the iteration count value in the particular register by one, and, as described above, reprocesses all the descriptors linked with the loop descriptor. In FIG. 2, descriptor channel block 108a stores the iteration count value of descriptor 203a in a register and every time
descriptor channel block 108a processes descriptor 203a, descriptor channel block 108a retrieves
the iteration count value stored in the register and determines whether the iteration count value is
zero or not. If the iteration count value is not zero, then descriptor channel block 108a processes
descriptor 203a, decrements the iteration count value in the register, and begins reprocessing
descriptor 201a. Therefore, each descriptor in FIG. 2 will be processed 5 times.

PROCESSING OF DESCRIPTORS BY DMAD

[0095] Once the descriptors 201a, 202a, 203a are programmed and stored in DMEM at
addresses 0x5000, 0x5010 and 0x5020, respectively, core processor 103a pushes the DMEM
addresses of descriptors 201a, 202a, 203a into a FIFO register accessible by DMAD 106a. In an
embodiment a core processor also indicates a particular hardware data channel of a descriptor
channel block of the DMAD that a descriptor should be pushed onto. The descriptor channel
block of the DMAD will either add the descriptors to an existing list maintained by the descriptor
channel block or build a new list. In building or adding to the list, the descriptor channel block
of the DMAD will write the second memory address pushed on to the FIFO register to the
descriptor's link address field of the first descriptor put on the list.

[0096] For example, in FIG. 2, memory addresses of descriptors 201a, 202a, 203a are pushed
on to a FIFO register accessible by core processor 103a and DMAD 106a. The memory address
of descriptor 201a is pushed in first, then the memory address of 202a is pushed in, and then the
memory address of 203a is pushed in. As described above, each descriptor channel block of a
DMAD maintains two lists per hardware data channel, an active list and a free list. Descriptor
cchannel block 108a determines whether the active list of the hardware data channel onto which
the memory addresses of descriptors 201a, 202a, 203a were pushed is empty. In an embodiment,
a descriptor channel block may determine whether a list is empty based on a counter associated
with the list. Descriptor channel block 108a determines whether the active list is empty based on
whether a counter associated with that active list is zero.

[0097] The descriptor channel block 108a adds to the active list by writing or copying the
memory address of descriptor 202a into the link address field of descriptor 201a and the memory
address of descriptor 203a into the link address field of descriptor 202a. If the active list of the
hardware data channel controlled by descriptor channel block 108a is not empty, then descriptor
channel block 108a writes or copies the memory address of descriptor 201a into the link address
field of the last descriptor that was pushed onto that particular hardware data channel, before
descriptor 201a. If the active list of the hardware data channel is empty, then descriptor channel block 108a copies the memory address of descriptor 201a from the FIFO register to a register designated to store the next descriptor that will be processed.

[0098] Descriptor channel block 108a retrieves the descriptor data of descriptor 201a from the DMEM using the memory address of descriptor 201a. Descriptor channel block 108a determines whether a wait condition needs to be satisfied by checking the WAIT field of the descriptor. In FIG. 2, a WAIT condition is required to be satisfied and is controlled by the value of EventO. Descriptor channel block 108a determines whether the wait condition is satisfied by determining the value of EventO. In an embodiment, a descriptor channel block may determine the value of an event by checking a register comprising values of all events that the descriptor channel block may use or is programmed to use. A WAIT condition is satisfied if the event specified by the WAIT field is at the specified WAIT value. Once, the WAIT condition is satisfied, a descriptor channel block does not have to wait any longer to process the descriptor.

[0099] In response to determining that the WAIT condition is satisfied, descriptor channel block 108a continues with the processing of the descriptor and determines the type of the descriptor. In an embodiment, the descriptor channel block determines the type of the descriptor based on an encoded value in the "Desc Type" field. In FIG. 2, descriptor 201a is a read descriptor where the source memory is the DDR memory and the destination memory is the DMEM local to core processor 103a or DMEM unit 102a. Descriptor channel block 108a transmits the control information of descriptor 201a, including the value for the "Src Auto Inc Allow" field, the source counter value, since the "Src Auto Inc Allow" field value is set, and any other information that may be needed to perform any of the operations specified in the descriptor to the read descriptor interface of DMAD 106a. In an embodiment, descriptor channel block 108a may transmit fragments of the control information of a descriptor and any other data needed to perform any of the other operations specified in the descriptor to the read descriptor interface of DMAD 106a. The size of each of the fragments depends on the width of the bus interface connecting descriptor channel block 108a with the read descriptor interface of DMAD 106a. Descriptor channel block 108a also determines whether the source address specified in the descriptor needs to be adjusted or modified based on the "Src Auto Inc Allow" field.

[0100] A descriptor channel block may use one or more values of the fields of the descriptor to determine whether or not the source address needs to be automatically adjusted or modified. In
FIG. 2 descriptor channel block 108a determines that the source address needs to be automatically adjusted or modified based on the value of the "Src Auto Inc Allow" field. Descriptor channel block 108a also determines whether the value of the counter specified in the "Counter Inc" field should be incremented based on the value of the "Src Addr Inc" field. The counter specified by the "Counter Inc" field of descriptor 201a is the "src" counter. Thus, descriptor channel block 108a also transmits to the read interface of DMAD 106a, the counter value of the "src" counter. Each hardware data channel is associated with a particular counter used in adjusting or modifying a source address, which is stored in a register and a managed by the descriptor channel block controlling the hardware data channel. The "src" counter in example of FIG. 2 is the counter associated with the hardware data channel controlled by descriptor channel block 108a. In FIG. 2, the "src" counter value is zero the first time it is transmitted. Descriptor channel block 108a, after transmitting the value of the "src" counter to the read descriptor interface of DMAD 106a, increments the value of "src" counter by 1.

[0101] A descriptor channel block is also associated with a destination counter value. The destination counter value is incremented in a similar manner as the source counter value, except that the value of the "Dest Addr Inc" field determines whether the destination counter will be incremented and the value of the "Dest Auto Inc Allow" field determines whether the destination address specified in the descriptor should be modified by an offset value. The "Counter Inc" field will specify or indicate the destination counter associated with the descriptor channel block.

PROCESSING OF DESCRIPTOR BY DMAX

[0102] Descriptor 201a data is transmitted to the arbitration unit 112a and then stored in the FIFO register 112b. FIFO register 112b then transmits the data to read descriptor parser logic block 121b and then into a register within the read descriptor parser logic block 121b. In an embodiment, if descriptor data is transmitted in fragments, then a descriptor parser logic block reassembles the descriptor data.

PROCESSING OF DESCRIPTOR BY DMAC

[0103] Descriptor read descriptor parser logic block 121b determines whether the source address specified in the descriptor data should be adjusted based on a value corresponding to an auto increment of source address field. In FIG. 2, "Src Auto Inc Allow" is such a field and based on the value of that field, read descriptor parser logic block 121b determines that the source
address should be automatically adjusted or modified. In an embodiment, source address may be automatically adjusted or modified according to the following formula:

\[ \text{New Source Address} = \text{source address} + \text{source address counter value} \times \text{rows} \times \text{column width (size in bytes)} \]

The source address above is the source address transmitted in the descriptor data. The counter value is the value of the counter that was transmitted along with the descriptor data, which in example of FIG. 2 is zero. The rows are the number of rows specified in the descriptor data and column width is the size of the column in bits. Therefore, plugging corresponding values into the above formula results in:

\[ \begin{align*}
\text{[0106]} & \quad = 0x0 \ 0000 \ 0000 + 0 \times 1000 \times 8 \\
\text{[0107]} & \quad = 0x0 \ 0000 \ 0000
\end{align*} \]

The New Source Address above is the same as the source address specified in the descriptor data since the value of the counter that helps determine the offset from the source address is zero. This is an accurate result because descriptor 201a represents the first 1000 rows of the 10,000 rows of data that are required to be processed in FIG. 2, therefore, an offset from the initially specified address is not required. Read descriptor parser logic block 121b transmits the control information of descriptor 201a and any other data required to perform any operations specified in descriptor 201a to an appropriate load engine block of data movement engine 130 based on the direction of the data movement indicated in the descriptor type field of a descriptor. In FIG. 2, the descriptor type field, "Desc Type", indicates that the data movement is from DDR to DMEM, therefore, the appropriate load engine block to which the control information of descriptor 201a and any other data required to perform any operations specified in descriptor 201a is transmitted is a DDR load engine block of data movement engine 130a.

The appropriate load engine block of a data movement engine determines the type of the source memory specified in the descriptor data and generates a read request based on the type of source memory. In FIG. 2, in response to determining that the source memory is DDR memory, the DDR load engine block of data movement engine 130a generates a read request to the system bus interface master block 123. In an embodiment, the amount of data requested in a read request may have a maximum threshold, and the number of read requests generated by an appropriate load engine block of a data movement engine may be based partly on the maximum threshold on the amount data that can be requested within one read request and the amount of...
data that a descriptor is requesting for in iteration of its processing. For example, if the maximum threshold is 256 bytes, then the appropriate load engine block of data movement engine 130a will generate 32 read requests to satisfy the 8000 bytes of data requested by descriptor 201a.

System bus interface master block 123 accepts the read requests and transmits the read requests to the appropriate memory unit interface that can accept the requests. In response to the requests, data corresponding to the read requests are returned to the system bus interface master block 123. Data corresponding to the read requests is transmitted from the system bus interface master block 123 to the appropriate load engine of the data movement engine that initiated the read requests. The appropriate load engine of the data movement engine transmits the data to an appropriate storage block unit within the data movement engine based on the destination memory unit indicated in the descriptor type field of the descriptor. In the example of FIG. 2, system bus interface master block 123 transmits the data to the DDR load engine of data movement engine 130a, and the DDR load engine transmits the data to the DMEM storage block unit within data movement engine 130a. DMEM storage block unit within data movement engine 130a transmits the data and control information of the descriptor to FIFO register 114b within DMAX 110a along with the destination address specified in descriptor 201a and an identifier associated with core processor 103a. FIFO register 114b identifies, using the core processor identifier included in the control information transmitted to FIFO register 114b from the DMEM storage block unit, DMEM unit 102a as the DMEM unit associated with the core processor identifier, and transmits data from write interface 114a to DMEM unit 102a via DMEM interface block 107a within DMAD 106a to store the data at the destination address specified in the descriptor.

DATA MANIPULATION OPERATIONS

In an embodiment, descriptor 201a may indicate that one or more tabular data manipulation operations, such as a gather, a scatter, a gather-scatter, or striding operations, may need to be performed on the tabular data requested from the source memory location.

GATHER OPERATION

A gather operation is a type of filtering operation wherein transfer of a row within a column of tabular data is dependent upon whether the particular row satisfies filtering criteria.

A bit vector or a list of row identifiers are used to evaluate whether the particular row satisfies the filtering criteria. In some embodiments, a high bit within a bit vector indicates that
the corresponding row within the column of tabular data should be transferred to the destination memory location. For example, if the bit vector is 1011, then the first, third and fourth rows within the column of tabular data should be transferred to the destination memory location. In some embodiments, a list of row identifiers may point to the exact rows within the column of tabular data that can be transferred to the destination memory location. Therefore, filtering of tabular data from a source memory location is performed prior to storing the tabular data in a destination memory location.

SCATTER OPERATION

[0114] A scatter operation allows for tabular data to be stored at non-linear destination addresses. In an embodiment, a scatter operation uses a bit vector or a list of row identifiers and the base destination address to determine the destination address for each row of column of tabular data being transferred to the destination memory location. A gather and a scatter operation, referred herein as a gather-scatter operation, may also be performed in combination.

GATHER-SCATTER OPERATION

[0115] In a gather-scatter operation, tabular data from the source memory location is filtered while the data is in-flight to the destination address, as described above, and the resulting data is stored within the destination memory in non-linear addresses. In an embodiment, use of a gather and a scatter operation ensures that each row of column of tabular data that satisfied the filtering criteria is stored in the destination memory in a manner such that the memory address offset between one filtered row of column of tabular data to another filtered row of column of tabular data mirrors the source memory address offset.

STRIDE OPERATION

[0116] A striding operation allows for tabular data to be converted from a column major format in a source memory to a row major format in a destination memory. The striding operation also allows for converting data from row major format in the destination memory to column major format in the source memory. To perform a conversion from column major to row major, a stride is determined based on the number of bytes per row of all interested columns of tabular data. The interested columns of tabular data will belong to a particular data table.

[0117] The data movement system, described herein, determines the destination address for each row of a column of tabular data using the base destination address specified in the descriptor and the stride. To perform a conversion from row major in destination memory in
order to store it in column major format in source memory, the data movement system uses the 
stride to determine all the rows of a column of tabular data. For example, to determine the 
second row of a column of tabular data, the data movement system uses the stride amount to 
offset from the address of the first row of column of data stored in the destination memory in a 
Row Major format. Thus the particular format in which tabular data is stored in a source memory 
will not provide compatibility issues in processing that data.

TRANSMISSION OF TABULAR DATA TO DESTINATION MEMORY

[0118] After transmitting data to FIFO register 114b, the DMEM store unit in data 
movement engine 130a transmits the descriptor return path of descriptor 201a to descriptor 
return block in DMAC 140, wherein descriptor return path of a descriptor includes a DMAD 
identifier, a descriptor channel block identifier and a descriptor identifier associated with the 
descriptor. Descriptor return block in DMAC 140 transmits the descriptor return path of 
descriptor 201a to a returned descriptor FIFO register within FIFO register 114b. FIFO register 
114b transmits the descriptor return path of descriptor 201a to descriptor channel block 108a in 
DMAD 106a. Descriptor channel block 108a sets the notify event to signal to the core processor 
103a that data requested by descriptor 201a is ready to be processed. In FIG. 2, the notify event 
is identified by the "Notify" field of descriptor 201a, and is EventO.

DESCRIPTOR RETURN PATH

[0119] Prior to processing a descriptor, a descriptor channel block determines whether there 
are any free or available descriptor identifiers that can be associated with the descriptor. If the 
descriptor channel block determines that no descriptor identifiers are available, then the 
descriptor channel block waits until a descriptor identifier is available. In an embodiment, one or 
more registers comprise all free descriptor identifiers. Once a descriptor identifier is available, 
the descriptor channel block begins processing a descriptor and associates the available 
descriptor identifier with the descriptor. In some embodiments the descriptor identifier may be a 
2 bit identifier, therefore, each descriptor channel block within a DMAD may process and 
support up to four different descriptors at a time. The descriptor identifier associated with the 
descrptor is included within the control information of that descriptor and transmitted to the 
DMAC.

[0120] Additionally, the descriptor channel block includes its own identifier within the 
control information transmitted to the DMAC. In some embodiments, the descriptor channel
block identifier is a one bit identifier. For example a value of zero in that bit identifies one
descriptor channel block of a DMAD and a value of one in that bit identifies the other descriptor
channel block of the DMAD. Descriptor channel block also includes a DMAD identifier
associated with the DMAD within which the descriptor channel block resides. In some
embodiments, the DMAD identifier may be 5 bits.

[0121] The descriptor identifier, the descriptor channel block identifier and the DMAD
identifier is collectively referred to herein as the descriptor return path. In some
embodiments, the identifier associated with the descriptor may be a sequence of bits, and different subsets of
the sequence of bits correspond to the descriptor identifier, the descriptor channel block
identifier, and the DMAD identifier. For example, the identifier associated with the descriptor
may be a sequence of 8 bits, where the five most significant bits correspond to the DMAD
identifier, the next bit corresponds to the descriptor channel block identifier and the least
significant two bits correspond to the descriptor identifier.

[0122] Once the DMAC completes processing all the operations necessary to satisfy the data
requests specified in a descriptor, then the DMAC configures a descriptor return path for that
descriptor. The descriptor return path of that descriptor includes the DMAD identifier, the
descriptor channel block identifier and the descriptor identifier associated with the descriptor and
included in the control information of that descriptor. The DMAC transmits the descriptor return
to its originating DMAD via the DMAX associated with the originating DMAD. A routing unit
within the DMAX associated with the originating DMAD determines the originating DMAD
based on the DMAD identifier included in the descriptor return path and transmits the descriptor
return path to the descriptor return interface block of the originating DMAD.

[0123] The descriptor return interface block of the originating DMAD determines the
descriptor channel block that processed that descriptor based on the descriptor channel block
identifier and transmits the descriptor return path to the descriptor channel block that processed
that descriptor. The descriptor channel block that processed that descriptor determines, based on
the descriptor identifier, the DMEM location of that descriptor. In an embodiment, the
association between a DMEM location of a descriptor and the descriptor identifier associated
with the descriptor is stored in a lookup table by the descriptor channel block that processed the
descriptor.
Descriptor channel block 108a determines whether the loop count is zero, and if the loop count is zero, the descriptor channel block 108a determines whether descriptor 201a will be added to the free list of the hardware data channel controlled by descriptor channel block 108a. In an embodiment, the descriptor channel block 108a may determine whether or not a descriptor will be added to the free list based on whether data of the descriptor indicates that the particular descriptor should be added to the free list. For example, there may be a free push field within the data of the descriptor that may have a value of 1 or 0 to indicate that the particular descriptor should be added to the free list. Additionally, if the loop count is zero, then the descriptor channel block 108a also decrements the active count value of that channel by 1. If the loop count is not zero, then descriptor channel block 108a terminates the processing of descriptor 201a for this particular iteration.

TRaversing the Hardware Managed List

After descriptor channel block 108a transmits descriptor 201a data to arbitration unit 112a, descriptor channel block 108a determines DMEM unit 102a address of the next descriptor within the active list of the descriptor channel block based on the link address field within the data of descriptor 201a. Descriptor channel block 108a retrieves data of descriptor 202a from DMEM 102a address 0x5010. Descriptor channel block 108a processes descriptor 202a similarly to how descriptor 201a is processed and transmits descriptor 202a data to arbitration unit 112a along with the "src" counter value, as indicated by descriptor 202a. The "src" counter value, when transmitted with the data of 202a, is 1. After descriptor 202a data is transmitted to arbitration unit 112a, descriptor channel block 108a increments "src" counter value by 1. Thus, the "src" counter value is now 2.

Descriptor 202a control information and the "src" counter value transmitted along with descriptor 202a data is stored in FIFO register 112b and then transmitted to read descriptor arbitration unit 121a and stored in read descriptor parser logic block 121b. Read descriptor parser logic block 121b again determines, similar to the way described above, that the source address specified in the descriptor data should be adjusted or auto incremented. Read descriptor parser logic block may adjust or modify the source address according to the same formula described above, which is:

New Source Address = source address + source counter value*rows*column width (size in bytes)
The source address above is the source address transmitted in descriptor 202a data. The counter value is the value of the counter that was transmitted along with descriptor 202a data, which is 1. The rows are the number of rows specified in descriptor 202a data and column width is the size of the column in bytes. Therefore, plugging the corresponding values into the above formula results in:

\[ = 0x0 \ 0000 \ 0000 + 1 \times 1000 \times 8 \]

\[ = 0x0 \ 0000 \ 1F40 \]

The New Source Address is now \( 0x0 \ 0000 \ 1F40 \), wherein \( 1F40 \) is the hexadecimal value equivalent to 8000. This is an accurate result because descriptor 202a represents the second 1000 rows of the 10,000 rows of data that are required to be processed in FIG. 2, therefore, an offset from the initially specified address is required. Read descriptor parser logic block 121b transmits the descriptor data to the appropriate load engine block of data movement engine 130a. The appropriate load engine block of data movement engine 130a processes descriptor 202a data similar to the processing of descriptor 201a data. Data retrieved for descriptor 202a is stored in DMEM 102a at the destination memory address specified in descriptor 202a.

DMEM 102a address of descriptor 202a is transmitted from descriptor return block of DMAC 140 to descriptor channel block 108a of DMAD 106a similar to the way DMEM 102a address of descriptor 201a was transmitted above. Descriptor channel block 108a sets notify event to signal to the core processor 103a that data requested by descriptor 202a is ready to be processed. In FIG. 2, the notify event identified by the "Notify" field of descriptor 202a is Event1. Descriptor channel block 108a again determines whether the loop count is zero and if it is zero, then descriptor channel block 108a completes processing of descriptor similar to the way described above for descriptor 201a. If the loop count is not zero, then descriptor channel block 108a terminates the processing of descriptor 202a for this particular iteration.

HARDWARE IMPLEMENTED LOOPING MECHANISM

After descriptor channel block 108a transmits descriptor 202a data to arbitration unit 112a, descriptor channel block 108a determines DMEM 102a address of the next descriptor within the active list of the descriptor channel block based on the link address field of descriptor 202a. Descriptor channel block 108a retrieves data of descriptor 203a from DMEM 102a address 0x5020. Descriptor channel block 108a determines that descriptor 203a is a program or control type descriptor and in particular a loop type descriptor. Descriptor channel block 108a
determines whether the loop count or iteration count is zero and if it is not zero, then descriptor channel block 108a decrements the loop count value by 1. As described above, the loop or iteration count is stored in a particular register designated for storing loop count values and the descriptor channel block retrieves the loop count value from the particular register and determines whether the loop count value is zero or not. If the loop count is not zero, then the descriptor channel block updates the loop count by decrementing the loop count value by 1 and stores the updated loop count value in the particular register. In FIG. 2, the iteration count value starts at 4, therefore, the first time descriptor 203a is processed, the loop or iteration count is 4 and after it is decremented by descriptor channel block 108a, the loop or iteration count will be 3.

[0134] In response to determining that the loop or iteration count is not zero, descriptor channel block 108a determines which descriptor it should loop back to and reprocess. Descriptor channel block 108a determines which descriptor it should loop back to based on descriptor 203a specifying a loop back address that identifies the DMEM address of the descriptor that should be processed again. In FIG. 2, the loop back address is specified in the "Loop Addr" field of descriptor 203a and DMEM 102a address of 0x5000 is the address of descriptor 201a within the DMEM 102a.

[0135] Descriptor channel block 108a retrieves descriptor 201a data from DMEM 102a. Descriptor channel block 108a determines whether the wait condition of descriptor 201a is satisfied based on value of event EventO. In FIG. 2, the wait condition of descriptor 201a is satisfied if EventO value is 0. As described above, descriptor channel block 108a previously set the value of EventO to 1 in order to indicate to core processor 103a that data is available for processing at the destination address specified by descriptor 201a. Therefore, if core processor 103a did not complete its processing of the data at the destination address specified by descriptor 201a, then the core processor will not clear EventO to 0, hence descriptor channel block 108a must wait until the EventO value is cleared to 0, i.e. set to 0.

[0136] If EventO value is cleared to 0, then descriptor channel block 108a processes descriptor 201a similar to the way descriptor 201a was processed in the previous iteration, described above. Descriptor channel block 108a continues to traverse the active list of the hardware data channel and based on the link address data of descriptor 201a retrieves descriptor 202a data. Descriptor channel block 108a processes descriptor 202a similar to the way it was
processed in the previous iteration. Descriptor channel block 108a continues to traverse the active list of the hardware data channel and based on the link address data of descriptor 202a retrieves data of descriptor 203a.

[0137] Descriptor channel block 108a again determines whether the loop count is 0. Loop count value is 3, therefore, descriptor channel block 108a decrements the loop count value by 1 and again loops back to descriptor 201a and processes descriptor 201a again. Descriptor channel block 108a continues to loop through the descriptors until the loop count value is 0. When the loop count value is 0, descriptor channel block 108a determines whether source or destination counters of the descriptor channel block 108a should be reset. In FIG. 2, descriptor 203a specifies, with "Src Count Reset" and "Dest Count Reset", fields that source and destination counters of hardware data channel must be reset.

[0138] Descriptor channel block 108a notifies core processor 103a that the loop has terminated or finished by setting the value of Event30 to 1 since Event30 is specified in descriptor 203a as the event that the core processor is configured to receive a notification from for descriptor 203a. Descriptor channel block 108a also decrements the active list count of the hardware data channel by 1. Processing of descriptor 203a is now complete and consequently processing of descriptors 201a and 202a. Thus, 10,000 rows of data from a source memory location have been processed by the data movement system.

MOVING A DATA TABLE USING DESCRIPTORS

[0139] FIG. 3 illustrates an example of moving an entire data table, comprising four columns, from a source memory to a local DMEM using descriptors. FIG. 3 comprises four data descriptors, 310a, 311a, 312a, 313a and one control descriptor 314a. Each descriptor represents a column of data of the data table stored in the source memory. Source memory in FIG. 3 is a DDR memory external to the data movement system. Destination memory in FIG. 3 is a DMEM, local to the core processor that programmed descriptors 310a, 311a, 312a, 313a. For the purpose of illustrating a clear example, FIG. 3 will be described using the hardware components and processes described in FIG. 1 and FIG. 2. For the purpose of illustrating a clear example, descriptors in FIG. 3 are also programmed by core processor 103a.

[0140] Descriptor 310a specifies the starting address of the first column of the data table in the source memory at the "Src Addr" field of descriptor 310a. Similarly, descriptors 311a, 312a, 313a, specify starting addresses of the second, third and fourth columns of the data table in the
source memory as their respective source addresses. Within the "Dest Addr" field, each of the descriptors 310a, 311a, 312a, 313a specify a different destination address within DMEM 102a, the DMEM local to core processor 103a. Descriptor 314a is a loop descriptor which indicates that the descriptor channel block that processes these descriptors should loop back to descriptor 310a, the descriptor that is stored at the DMEM 102a address specified in descriptor 314a.

[0141] One of the descriptor channel blocks of DMAD 106a will add descriptors 310a, 311a, 312a, 313a, 314a to the active list of the hardware data channel controlled by the descriptor channel block similar to the way descriptor channel block in FIG. 2 added descriptors to the active list. The descriptor channel block will begin processing descriptors of FIG. 3 with descriptor 310a. Processing of descriptor 310a is similar to the way descriptor processing has been described in FIG. 2 and FIG. 1.

[0142] However, unlike FIG. 2, the source address counter of the descriptor channel block will not be incremented after descriptor 310a data is transmitted to the arbitration unit 112a because descriptor 310a does not specify that the source address counter should be incremented. This is accurate for FIG. 3 because source addresses of descriptors 311a, 312a, and 313a are different for each since each descriptor is processing a different column of data of the data table, each of the descriptors 311a, 312a, and 313a, in the first iteration of the loop, should process their respective column of data from the source address specified in the descriptor, without any offset from the source address. Similarly, descriptor 311a, 312a also do not specify that source address should be incremented in order to ensure that the descriptor following them begins processing their respective columns of data from the correct memory address.

[0143] Descriptor 313a data specifies that the source address counter should be incremented. Thus, the source counter of descriptor channel block 108a is incremented after control information of descriptor 313a is transmitted to arbitration unit 112a. Incrementing source address counter after the last bytes of the control information of descriptor 313a is transmitted ensures that the next time descriptors 310a, 311a, 312a, 313a are processed the source address specified in 310a, 311a, 312a, 313a are adjusted, modified or offset accurately. The remaining aspects of processing and returning descriptors 310a, 311a, 312a, 313a, 314a are similar to the way it is described in FIG. 1 and FIG. 2.

[0144] Thus, data movement system may copy an entire data table from a source memory and store the data table in a destination memory using descriptors.
PERFORMING DATA MANIPULATION OPERATIONS

[0145] Approaches for performing tabular data manipulation operations while moving data from a source memory location to a destination memory location are initially illustrated using a gather operation. As described earlier, a gather operation is a type of filtering operation wherein transfer of a row within a column of tabular data is dependent upon whether the particular row satisfies a filtering criteria and a scatter operation allows for tabular data to be stored at non-linear destination addresses. Other types of tabular data manipulations operations will be described later.

[0146] FIG. 4 illustrates example descriptors used for performing a gather operation. These descriptors include a data descriptor that specifies to perform a gather operation for a column. A data descriptor may also specify a scatter operation. A gather or scatter operation can be performed using several filter modes for filtering rows, one mode using a list of row identifiers (RID) and one using a bit vector (BV).

[0147] The filter mode is specified by a value in a "scat/gat" register of a DMAD. A program descriptor is used to configure the value of a "scat/gat register". The value of the scat/gat register specifies not only whether the filter mode is RID or BV, but also the storage address of an RID or BV within the DMS memory. A RID or BV that is used to perform a gather operation is copied from DMEM to DMS memory, or from main memory to DMS memory, using a data descriptor.

[0148] Referring to FIG. 4, it depicts four descriptors, program descriptor 401a, data descriptor 402a and data descriptor 403a, and loop descriptor 404a. For the purposes of illustration, FIG. 4 will be described using the hardware components and processes described in FIG. 1, FIG. 2, and FIG. 3; descriptors in FIG. 4 are also programmed by core processor 103a. The descriptors are used to perform a gather operation using a BV.

[0149] In FIG. 4, core processor 103a configures program descriptor 401a, which is used to configure a scat/gat register. The value of the scat/gat register specifies various aspects of performing a gather operation. Descriptor 401a indicates that it is a program descriptor by specifying in the descriptor type field a value that a DMAD decodes to be corresponding to a program descriptor. Descriptor 401a specifies to configure the scat/gat register value by specifying the scat/gat register in the "Register" field. The value of the scat/gat register specifies
whether RID or BV mode is being used for filtering, and if RID mode is specified, what the width of a RID is.

Accordingly, Descriptor 401a specifies whether RID or BV mode is being used for filtering by specifying a particular value in the "Mode" field. In descriptor 401a, that value specifies BV mode. If the mode had been RID, then descriptor 401a should have specified the width of a RID. According to an embodiment, the width of a RID may be 1, 2, 4, or 8 bytes.

Descriptor 401a also specifies the DMS memory address of where the BV should reside. The BV may be initially formed and/or stored by core processor 103a in its DMEM unit 102a. A separate descriptor, descriptor 402a, specifies to move a RID or BV to this DMS memory address.

Descriptor 402a is the data descriptor in FIG. 4 that specifies to move the BV from DMEM to DMS memory. Descriptor 402a specifies that the BV is stored at the DMEM location specified by the "DMEM Addr" field, i.e. the source memory location. Descriptor 402a also specifies, using the "DMS Addr" field, the destination address of where that BV is to be stored. The value in the "DMS Addr" field specifies a memory location within DMS memory 150 to store the BV. Moving or copying and storing the BV (or RID) to a DMS memory reduces latency in accessing the BV (or RID) by the data movement engine while performing the gather operation. In an embodiment, Descriptor 402a can specify to move a BV (or RID) from main memory or a DDR memory unit to DMS memory.

Furthermore, descriptor 402a also indicates the length of the BV that is to be stored in the destination memory specified in program descriptor 401a and 402a. In an embodiment, the length of the BV that is to be stored is based on the number of rows specified in the particular descriptor that will be processed to move the BV to DMS memory from DMEM. Descriptor 402a also specifies in the "Rows" field the number of bits that particular bit vector comprises.

In FIG. 4, descriptor 403a is a data descriptor that specifies a gather operation to be performed on a column of tabular data. Descriptor 403a specifies within the "Gather" field that a gather operation should be performed. In an embodiment, the value within the "Gather" field may be an encoded value such as a bit or binary number that indicates to a descriptor channel block within a DMAD that a gather operation is being requested to be performed on the column of tabular data. The other fields of descriptor 403a are similar to the fields of the descriptors
described in FIG. 2 and FIG. 3, and similar to descriptors processed by the DMS shown in FIG. 1.

[0155] Descriptor 404a is a loop descriptor. It is configured and processed similarly as described for loop descriptor 314a.

PROCESSING GATHER OPERATION

[0156] In FIG. 4, elements 401b, 402b, 403b and 404b each correspond to operations performed for descriptors 401a, 402a, 403a, and 404a, respectively. The purpose of these elements is to depict the order of operations performed for descriptors 401a, 402a, 403a, and 404a.

[0157] Core processor 103a configures descriptors 401a, 402a, 403a, and 404a, and transmits the respective memory locations within DMEM unit 102a to one of descriptor channel blocks in DMAD 106a. For example, core processor 103a may transmit an identifier associated with descriptor channel block 108a along with the memory location of descriptor 401a to a FIFO interface within DMAD 106a. DMAD 106a, based on the identifier received, adds descriptor 401a to the active list of descriptor channel block 108a. The other descriptors 402a, 403a, and 404a are processed similarly. Retrieval of descriptors 401a, 402a, 403a, and 404a are performed in the same manner as described in relationship to FIG. 1, FIG. 2 and FIG. 3.

[0158] In FIG. 4, descriptor channel block 108a decodes descriptor 401a and identifies descriptor 401a as a program descriptor based on the descriptor type field of descriptor 401a. Since the descriptor 401a identifies the register to configure as the scat/gat register, the descriptor channel block 108a configures scat/register of DMAD 106a to a value that specifies that the filtering mode is BV, and the address in DMS 150 of where the BV will be stored.

[0159] In FIG. 4, descriptor channel block 108a decodes descriptor 402a and identifies descriptor 402a as a data descriptor based on the descriptor type field of descriptor 402a. As described above, the descriptor type field of a data descriptor also indicates the direction of data movement, thus identifying the source memory location, from where data is to be moved or copied from, and a destination memory location, to where the data is to be stored. Descriptor type field of descriptor 402a indicates that the direction of data movement is from DMEM to a memory unit in DMS memory, therefore, the memory location specified in the "DMEM addr" field is the source memory location of data within the DMEM unit 102a and the memory location...
specified in the "DMS addr" field is the destination memory location within DMS memory unit used for storing the BV.

Routing unit 114a transmits the read request, based on the control information of descriptor 402a, to the DMEM interface block 107a. DMEM interface block 107a retrieves the BV from DMEM unit 102a, based on the memory location specified in the "DMEM Addr" field of descriptor 402a. The control information received at DMEM interface block 107a includes the memory location specified in the "DMEM Addr" field of descriptor 402a. DMEM interface block 107a generates a read response comprising the BV from DMEM unit 102a and control information of descriptor 402a. DMEM interface block 107a transmits the read response to arbitration unit 113a. Arbitration unit 113a, based on the control information of descriptor 402a, transmits the read response to data movement engine 130a.

Data movement engine 130a stores the BV at the memory location specified in the "DMS Addr" field of descriptor 402a in the memory unit of the DMS used for storing row identifiers or BVs.

Processing of descriptors that specify data manipulation operations that use the BV and that follows a descriptor that stores the BV in a memory unit in DMS may be temporarily suspended until all of the required BV is successfully stored in the memory unit in the DMS, thereby preventing generation of read or write requests using an incomplete BV for the descriptors that specify the data manipulation operations.

The descriptor channel block may temporarily suspend processing of descriptors that follow the current descriptor that is being processed, based on the type of wait event and wait conditions specified in the current descriptor. One type of wait event, referred to herein as a "pre-wait" event, is a wait event that indicates to the descriptor channel block that the processing of the descriptor that specified that event type should not begin until the wait condition, also specified in that descriptor, is satisfied. An example of this type of wait event is described in FIG. 2 and FIG. 3 above. Another type of wait event, referred to herein as "post-wait" event, is a wait event that indicates to the descriptor channel block that the descriptor specifying the post-wait event type can be processed, however that the descriptor channel block cannot begin processing the next descriptor in the active list of that descriptor channel block until the wait condition specified in the descriptor with the post-wait is satisfied and that the operation the descriptor is performing has completed.
FIG. 4 illustrates the use of the post-wait event to temporarily suspend processing of the other descriptors in the active list of the descriptor channel block. In FIG. 4, descriptor 402a specifies that the event type is a post-wait event, therefore descriptor channel block 108a, the descriptor channel block processing descriptor 402a, temporarily suspends processing of the next descriptor until the post-wait condition specified in descriptor 402a is satisfied. As described above, a post-wait condition is satisfied if the event specified in the "WAIT" field of the descriptor is at the value specified in the "WAIT" field of the descriptor and the operation specified by the descriptor has been completed. In FIG. 4, the post-wait condition of descriptor 402a is satisfied when the value of Event4 is set to 1. Therefore, descriptor channel block 108a, the descriptor channel block processing descriptor 402a, does not begin processing of the next descriptor on the active list of descriptor channel block 108a, descriptor 402b, until the value of Event4 is set to 1. Thus, all descriptors after descriptor 402a in the active list of descriptor channel block 108a are temporarily suspended until the wait condition of the post-wait event of descriptor 402a is satisfied and the operation specified by descriptor 402a, movement of the BV from DMEM to DMS memory, has completed.

Once the wait condition of descriptor 402a is satisfied, descriptor channel block 108a traverses its active list as described in relationship with FIG. 2 and FIG. 3, and begins processing descriptor 403a. Descriptor channel block 108a determines, based on the descriptor type field, that descriptor 403a is a data descriptor and that the direction of data movement is from an external memory into DMEM unit 102a. Descriptor channel block 108a also determines that a gather operation should be performed on the tabular data being stored into the DMEM unit 102a.

As described above, the data movement system performs data manipulation operations, such as a gather operation on the tabular data, prior to that tabular data being stored in the destination memory unit. Therefore, the data movement engines in DMAC 140 will perform the gather operation requested by descriptor 403a on the tabular data retrieved from the source memory location prior to any of the tabular data being stored in the destination memory unit of the destination memory location specified in descriptor 403a.

Descriptor channel block 108a transmits the control information of descriptor 403a including the gather operation specified in descriptor 403a to arbitration unit 112a. Arbitration unit 112a, based on the descriptor type of descriptor 403a, transmits control information of the scat/gat register of DMAD 106a, the gather operation specified in descriptor 403a, and an
identifier associated with DMAX 110a to read descriptor parser block 121b via read descriptor arbitration unit 121a.

[0168] As described above, one or more data movement engines may be associated with one or more DMAX units such that the read descriptor or write descriptor parsers of a DMAC will transmit the descriptor control information including any data manipulation operations specified in the descriptor received from the one or more DMAX units to the associated data movement engines. In FIG. 1, data movement engine 130a is associated with DMAX 110a for most data movement descriptors, therefore, read descriptor parser 121b or write descriptor parser 120b transmit the descriptor control information including any data manipulation operations specified in the descriptor received from DMAX 110a to data movement engine 130a.

[0169] Read descriptor parser 121b, based on the received DMAX identifier, transmits control information from scat/gat register of DMAD 106a, the gather operation specified in descriptor 403a, and an identifier associated with DMAX 110a to data movement engine 130a since data movement engine 130a is associated with DMAX 110a.

[0170] A data movement engine determines that data manipulation operations are to be performed on the tabular data that is being moved or copied from the source memory location based on the descriptor information received from either a read descriptor parser or a write descriptor parser. In FIG. 1, data movement engine 130a, retrieves the BV stored in the memory unit of DMS, designated for storing the BV, at the memory location specified in the "DMS Addr" in control information from the scat/gat register as detailed in the sequence shown in FIG. 4. This control information also informs data movement engine 130a that the filtering mode is BV.

[0171] Data movement engine 130a parses the retrieved BV in order to determine whether a read request to the source memory should be generated. As described above, each bit within a bit vector corresponds to a row in the column of tabular data stored at the source memory location in the source memory unit. The value of each bit identifies whether the corresponding row in the column of tabular data may be stored in the destination memory unit or whether it may be filtered out, thus not stored in the destination memory unit. In an embodiment, a row in a column of tabular data will be stored in a destination memory unit if the value of the corresponding bit in the BV is a 1 and a row in a column of tabular data will be filtered out if the value of the corresponding bit in the BV is a 0.
In an embodiment, a data movement engine parses the BV to determine whether values of a sufficient number of consecutive bits indicate that their corresponding rows will be filtered out such that a read or write request for their corresponding rows is not generated. The data movement engine checks whether the total cumulative size of data from the corresponding rows exceeds a threshold size of data in determining whether a read or write request for the corresponding rows should be generated. In determining the total cumulative size of data from the corresponding rows, a data movement engine may calculate the cumulative size using the column width of the column of tabular data, specified in the descriptor and transmitted to the data movement engine, and the number of bits that indicate that their corresponding rows will be filtered out. By using a threshold size of data in part to determine whether a read or write request should be used allows the data movement system to provide a flexible process of determining read or write requests for a particular set of rows within a column of tabular data.

The threshold size of data may depend in part on the amount of data per read or write request that can be transmitted from a source memory location. For example, if the amount of data that can be transmitted per read or write request is 256 bytes, then the threshold amount of data may be set to 256 bytes and a data movement engine may not generate a read request if a consecutive number of bits that indicate that their corresponding rows should be filtered out multiplied by the column width equal or exceed 256 bytes. The data movement engine may skip the generation of a read request for the rows corresponding to that set of consecutive number of bits and generate read or write requests for the rows corresponding to the remaining number of bits in the bit vector. Consequently, the number of read or write requests generated by a data movement engine when a gather operation is specified depends on the consecutive number of bits that indicate that their corresponding rows will be filtered out and not simply on the number of rows specified in a descriptor.

In FIG. 4, descriptor 403a specifies the column width is 8 bytes, and if the threshold amount of data is 256 bytes, then data movement engine 130a may determine that a read request will not be generated if the values of 32 consecutive bits in the retrieved BV indicate that their corresponding rows in the column of tabular data should be filtered out. Thus, data movement engine 130a generates read requests based on the values of the bits in the BV data. Data movement engine 130a transmits the read requests and control information of descriptor 403a to
system bus interface master block 123. System bus interface master block 123 stores the control information of descriptor 403a in a register within system bus interface master block 123.

[0175] System bus interface master block 123 transmits the received read requests to the source memory unit. For each read request, system bus interface master block 123 receives data from source memory unit. System bus interface master block 123 transmits the column of tabular data from the source memory unit and the control information of descriptor 403a to data movement engine 130a. Data movement engine 130a, based at least in part on the received gather operation and/or control information of descriptor 403a from system bus interface master block 123 transmits the column of tabular data to a gather operation engine.

[0176] The gather operation engine, based on the BV, determines whether the corresponding row within the received column of tabular data should be stored in the destination memory unit. For each bit in the BV, the gather operation engine, based on the value of the bit, determines whether the corresponding row should be stored in the destination memory unit. The gather operation engine identifies the corresponding row of a bit within the column of tabular data based on the following:

[0177] Gather address = Descriptor source base address + (column width * bit number)

[0178] The gather address given above represents the address of the row corresponding to the bit number. The descriptor source base address is the source memory location specified in descriptor 403a. The column width is the size of the column in bytes and bit number represents a bit within the BV and bit number starts from zero. For example, if the BV is comprised of four bits, 101 1, then bit number zero is the first bit from the right and has value of one. Bit number one is the second bit from the right and has a value of one. Bit number two is the third bit from the right and has a value of zero. Bit number three is the fourth bit from the right and has a value of 1.

[0179] In an embodiment, the gather operation engine stores data gathered from source addresses where the values of the bits of bit numbers are 1. The gather operation engine transmits the data of the rows with corresponding bit values of 1 to the DMEM store engine. The gathered data is accumulated and stored in a contiguous memory range in the destination memory. The data of the rows is transmitted to DMEM unit 102a as described in relationship to FIG. 1, FIG. 2, and FIG. 3.

Scatter Operation
Similar to the gather operation, a descriptor may specify a scatter operation. Data received from a source memory location is transmitted to the scatter operation engine. The BV is also transmitted to the scatter operation engine. The scatter operation engine determines the destination address for each row of column of tabular data according to the following:

\[ \text{Scatter address} = \text{Descriptor destination base address} + (\text{column width} \times \text{bit number}) \]

The scatter address above represents the destination address of where each qualifying row of the column of tabular data will be stored. The descriptor destination base address is the address specified in the descriptor. The column width is the size of the column in bytes and bit number represents a bit within the BV where the bit number starts from zero.

A descriptor may specify, both, a gather and a scatter operation to indicate that a gather operation should be performed on the data from the source memory location and that a scatter operation should be performed on the data upon which a gather operation is performed. The functionality of the gather and the scatter operations when, both, a gather and a scatter operation are specified is similar to the way described above for the gather operation and for the scatter operation.

**Striding Operation**

A group of descriptors (one per column) may specify that a striding operation should be performed on a group of columns of tabular data from their source memory location to their destination memory location. The striding operation as described above transforms the columns of tabular data stored in column major format at the source memory location to row major format at the destination memory location or columns of tabular data stored in row major format at the source memory location to column major format at the destination memory location.

A core processor may determine a stride amount to help transform from a column major format to a row major format or vice versa. The stride amount is the number of bytes per row across all columns of tabular data that are of interest. For example, if two columns of tabular data are to be moved or copied from an external memory unit and stored into a DMEM unit, then the stride amount is the sum of the column widths of both columns of tabular data.

Data from the source memory location is transmitted to the stride operation engine and the stride operation engine transforms tabular data stored in a column major format to a row major format according to the following:

\[ \text{destination address} = \text{destination base address} + (\text{dst count} \times \text{stride}) \]
The destination address above represents the destination address in DMEM and the destination base address represents the destination address specified in the descriptor. Dst count represents the row number of the column element being moved and stride represents the stride amount in bytes. Thus each row of tabular data from the source memory stored in column major format will be stored at the destination memory in row major format.

The stride operation engine may transform tabular data stored in a row major format to a column major format according to the following:

source address = source base address + (src_count * stride)

The source address above represents the source address in DMEM, and the source base address represents the source address specified in the descriptor. Src count represents the row number of the column element being moved and stride represents the stride amount in bytes. Thus each row of tabular data stored in row major format at a source memory location will be stored at the destination memory in column major format.

Aligned Run Length Encoding (ARLE) Operation

The data movement system described herein allows for encoded or compressed data from a source memory location, such as main memory or a DDR memory unit, to be decompressed prior to storing the data in the destination memory unit, such as a DMEM unit, and for un-encoded or decompressed data from a source memory location, such as a DMEM unit, to be compressed prior to storing the data in main memory or a DDR memory unit.

Compression and decompression of tabular data is performed on the fly, based on values specified in certain fields of a descriptor. Compression and decompression of a column of tabular data are performed based on a corresponding run and data array. A run comprises one or more elements, where each element specifies how many times a corresponding data element of a data array is repeated. Compression takes an uncompressed column of data and generates a data array and a corresponding run array; decompression takes a run array and a data array and generates an uncompressed column of data. When decompressing ARLE compressed data, the DMS logic reads the data and run arrays from main memory and stores them in internal DMS memory buffers. Buffering run and data arrays stored in main memory in DMS memory avoids multiple accesses to the same array in main memory, thus reducing the overall power consumed by the DMS, and improving performance of decompressing tabular data within the DMS. Likewise, when performing ARLE compression, the DMS creates the run and data arrays in local
DMS memory and moves the arrays to main memory when the local memory fills. Using the local DMS memory for compression has benefits similar to those described for using local DMS memory for decompression.

[0195] The total size of a run and data array depends in part on the number of rows being moved or copied from the source memory location. The size of the arrays also depends in part on the width of the column of tabular data that is being moved or copied from the source memory location and the average compression ratio of the column of tabular data. The size of the run array depends on the maximum run value that can be stored. In one embodiment, a run element is one byte wide. For example, if the width of the column of tabular data is 2 bytes, the number of rows being moved of that column tabular data is 256, and the average compression ratio is 1:4, then the total size of the run array is 256/4 or 64 bytes and the total size of the data array is (256/4) * (width of the column of tabular data) or (256/4) * 2, or 128 bytes. When performing decompression, the run and data arrays are moved or copied from a source memory location to the local DMS memory unit coupled with the ARLE compression/decompression block using descriptors.

[0196] A core processor configures an auxiliary type descriptor to provide the address of the run array to the data movement system. The descriptor type field of the auxiliary descriptor comprises a value that indicates that the descriptor is an auxiliary type descriptor. In an embodiment, that value is a sequence of binary numbers that indicate to the descriptor channel block that the descriptor is an auxiliary type descriptor. The descriptor channel block decodes the auxiliary descriptor similarly to the methods described above. The descriptor channel block determines the control information for the auxiliary descriptor similarly to the methods described above.

[0197] The run address provided in the auxiliary descriptor is the start address of the run. In an embodiment, the auxiliary descriptor comprises a "Run Address" field and the run address is provided as the value of the "Run Address" field. In an embodiment, the width of the run array is limited to one byte. The descriptor channel block includes the descriptor type of the auxiliary descriptor and the run address within the control information determined by the descriptor channel block. The descriptor channel block transmits the control information of the auxiliary descriptor to a data movement engine similar to the methods described above.
Since the descriptor is an auxiliary descriptor, the data movement system does not move any data from a source memory location to a destination memory location. Instead, the data movement system stores the values provided in the auxiliary descriptor to use it in processing the data descriptor that follows the auxiliary descriptor. Thus, the data movement engine, based on the control information and in particular the descriptor type, determines that the descriptor is an auxiliary descriptor and stores the run address provided in the control information in a register and does not generate any read requests to the system master bus block interface to retrieve the run array from the run address location.

After configuring the auxiliary type descriptor, the very next descriptor configured by the core processor is a data descriptor. The information provided in the auxiliary type descriptor preceding the data descriptor will be used in processing the data descriptor. The data descriptor provides the starting address of the aligned run length encoded tabular data in a source memory unit, such as the main memory or another memory unit. This is what was referred to as the data array in the preceding paragraphs. The "Rows" field of this data descriptor comprises information about the number of rows that will result after the aligned run length encoded tabular data is decoded. The "Width" field of the data descriptor comprises a value indicating the column width of the data array available at the starting address of the data array, which is provided in the data descriptor.

The data descriptor following the auxiliary descriptor initiates the movement of the run array needed in run length aligned decoding of the tabular data. The data movement engine generates a read request for the run array where the source address included in the read request is the Run address from the auxiliary descriptor that was stored in a register accessible by the data movement engine. The data movement engine transmits the read request to the system bus interface master block. The data movement engine also generates a read request for the data array and transmits the read request to the system bus interface master block.

The system bus interface master block retrieves the run and data arrays from their respective source memory locations similar to the methods described above. The system bus interface master block transmits the arrays to the data movement engine. The data movement engine decodes the data array using the run array. The decoded run length aligned encoded data is stored in the destination memory location specified in the data descriptor similar to the methods described above. The destination memory location in one embodiment can either be in
DMEM associated with the originating descriptor channel block of the data descriptor (such as 102a in Figure 1) or in DMS memory to be used for subsequent DMS processing (150 in Figure 1).

DATA MANIPULATION OPERATION BLOCKS

[0202] FIG. 5 illustrates an example arrangement of data manipulation operation engines. For the purposes of illustration, a clear example of data manipulation operations are shown using elements of FIG. 1. FIG. 5 depicts a plurality of data manipulation operation blocks within each of data movement engines 130a, 130b, 130c, and 130d.

[0203] FIG. 5 comprises stride operation block 509, scatter operation block 510, gather operation block 511 and ARLE decompression block 512. As described above, system bus interface master block 123 forwards data requests to the source memory location and also receives data, including tabular data, from the source memory location. System bus interface master block transmits data received, in response to the read requests, from the source memory location and control information of the descriptor that requested the data to arbitration unit 516. Arbitration unit 516 comprises multiplexer 516a and arbiter 516b. Multiplexer unit 516a transmits data received from the source memory location and control information of a descriptor to routing unit 514.

[0204] Routing unit 514, based on the control information received from multiplexer 516a, transmits the data received from the multiplexer 516a to either one of the data manipulation operation blocks 509, 510, 511, 512 or to routing unit 502, comprising multiplexer 502a and routing controller 502b. For example, if control information received from multiplexer 516a does not indicate that any data manipulation operations are to be performed, then the tabular data and the control information received from multiplexer 516a is transmitted to multiplexer 502a. Therefore, the data movement system provides flexibility in skipping one or more data manipulation operation blocks rather than wasting clock cycles or data manipulation operation block resources when performance of data manipulation operations is not required.

[0205] Similarly, if control information received from multiplexer 516a indicates one or more data manipulation operations, then the tabular data and the control information received from multiplexer 516a is transmitted to the appropriate data manipulation operation block. Routing unit 514 may be configured with a particular order in which data manipulation operations are to be performed on a set of tabular data when the control information indicates
that more than one data manipulation operation is to be performed on the data received from multiplexer 516a.

[0206] An example order in which data manipulation operations are to be performed on a set of tabular data is, first, decompression of the tabular data by an ARLE operation block, if the control information associated with that tabular data indicates that such an operation is to be performed. Then, a gather operation is performed on the decompressed tabular data, if the control information indicates that a gather operation should be performed. Following the gather operation, a scatter operation is performed on the tabular data remaining after the gather operation, if the control information indicates that a scatter operation is to be performed. Finally, a stride operation is performed on the tabular data upon which the scatter operation is performed, if the control information indicates that a stride operation should be performed. This example order of data manipulation operations is illustrated in FIG. 5.

[0207] In FIG. 5, routing unit 514 is configured to route tabular data received from multiplexer 516a to ARLE decompression block 512 first, if control information received from multiplexer 516a indicated that the tabular data should be decompressed. ARLE decompression block 512 transmits the decompressed tabular data and the associated control information to routing unit 508. Routing unit 508, based on the control information, determines whether additional data manipulation operations are requested or required and transmits the data accordingly. Suppose the control information indicates that a gather operation should also be performed, then routing unit 508 transmits the tabular data resulting from the ARLE decompression and the control information to multiplexer 515c, which transmits it to gather operation block 511.

[0208] Gather operation block 511 performs the gather operation as described above and transmits the resulting tabular data and the control information to routing unit 507. Routing unit 507 determines if the control information indicates performance of any other data manipulation operations on the tabular data. In an embodiment, routing unit 507 only checks for whether the control information indicates that either a scatter or a stride operation is to be performed, since the next operations in the order are a scatter operation and/or a stride operation, but not an ARLE operation. Additionally, routing unit 507 can transmit to either the scatter operation block 510 or the stride operation block 509, but cannot transmit to ARLE decompression block 512.
Suppose that the control information indicated that a stride operation is to be performed, then routing unit 507 transmits the resulting tabular data and the control information to multiplexer 515a. Multiplexer 515a transmits the tabular data and the control information to stride operation block 509. Thus, scatter operation block 510 is completely skipped over, saving the resources of scatter operation block 510. Stride operation block 509 transmits the resulting tabular data to routing unit 505. Routing unit 505, based on the destination memory location indicated in the control information, transmits the resulting tabular data and the control information to either multiplexer 501a, or multiplexer 502a. Similarly, routing units 506, 507, 508, based on the destination memory location, may transmit data from their respective data manipulation operation blocks to multiplexer 501a or multiplexer 502a, if the control information indicated that no other data manipulation operation is requested or required.

Data transmitted for multiplexer 501a is destined for DMS memory. The data is initially aligned and accumulated in data alignment/accumulator 503, which writes the data to the destination memory location within DMS memory. Arbiter 501b is an arbiter for multiplexer 501a.

Data transmitted for multiplexer 502a is destined for DMEM. The data is aligned and accumulated in data alignment/accumulator 504, which writes the data to the destination memory location within DMEM memory. Arbiter 502b is an arbiter for multiplexer 502a.

Therefore, the output of each of the data manipulation operation blocks described above may be cascaded into an input of one of the other data manipulation operation blocks. For example, the output from the gather operation block can be fed in as the input into the scatter operation block. Similarly the output from ARLE compression/decompression block can be fed into the input of a gather, scatter, or a stride operation block. Furthermore, a data movement engine may skip some or all of the data manipulation operation blocks based on the control information provided by the descriptors.

Finally, the data manipulation operation blocks may each be concurrently executing (i.e. in the same clock cycles) on data manipulation operations on tabular data for different descriptors. For example, routing unit 514 routes tabular data received from multiplexer 516a for a first descriptor to ARLE decompression block 512. ARLE decompression block 512 decompresses the tabular data and transmits the decompressed tabular data and the associated control information for the descriptor to routing unit 508. Routing unit 508 transmits the tabular
data resulting from the ARLE decompression and the control information for the first descriptor to multiplexer 515c.

[0214] Multiplexer 515c transmits the tabular data to gather operation block 511. Meanwhile, routing unit 514 routes tabular data received from multiplexer 516a for a second descriptor to ARLE decompression block 512. ARLE decompression block 512 decompresses the tabular data for the second descriptor while (i.e. within the same clock cycles) gather operation block 511 performs a gather operation on the tabular data for the first descriptor.

METHOD OF MOVING DATA AND PERFORMING DATA MANIPULATION OPERATIONS WITHOUT EXECUTING SOFTWARE PROGRAMS

[0215] FIG. 6 illustrates an example process for moving data using hardware circuitry without execution of software. In an embodiment, the operations described for FIG. 6 may be performed by the data movement system described in FIG. 1.

[0216] In steps 601a and 601b, in response to a particular memory location being pushed into a first register within a first register space that is accessible by a first set of electronic circuits, said first set of electronic circuits accesses a descriptor stored at the particular memory location. In an embodiment, the particular memory location may be a memory address within a DMEM local to a core processor that programmed the descriptor. In an embodiment the first set of electronic circuits may be the group of electronic circuits of any DMAD described in FIG. 1.

[0217] In step 602, the first set of electronic circuits determine, based on the descriptor, control information indicating said one or more data manipulation operations to perform on the tabular data. In an embodiment, the descriptor may specify a source memory location and a destination memory location for said tabular data. In some embodiments, the source memory location may be within a main memory such as a DDR and the destination memory location may be within a DMEM as described in FIG. 1. In some embodiments, the source memory location may be within a DMEM and the destination memory location may be within a main memory. In an embodiment, the descriptor may specify a width of a column of tabular data, a number of rows of tabular data, and one or more data manipulation operations to perform on the column of data.

[0218] In step 603, the first set of electronic circuits transmit, using a hardware data channel, the control information to a second set of electronic circuits to perform the one or more data manipulation operations. In an embodiment, the second set of electronic circuits may be the
group of electronic circuits within the DMAC described in FIG. 1. In an embodiment, a single DMAC, as described in FIG. 1, may process instructions from a plurality of DMAD units described in FIG. 1. Therefore, because a single DMAC services a plurality of DMAD units, the data movement system described herein significantly reduces gate count, area and power necessary to move data.

[0219] In step 604a, according to the control information, said second set of electronic circuits retrieve said tabular data from a source memory location. In an embodiment, the source memory location may be specified by the descriptor. In step 604b, the second set of electronic circuits apply one or more data manipulation operations to said tabular data to generate a data manipulation result. In an embodiment, the data manipulation operations are performed prior to storing the tabular data in the destination memory location at a sustained high bandwidth into a format expected by the core processor. In step 604c, the second set of electronic circuits cause said data manipulation result to be stored at a destination location. In an embodiment, the destination location may be specified by said descriptor.

PARTITIONING

[0220] Partitioning of rows is performed in three stages, which are referred to herein as partitioning stages. The partitioning stages are illustrated herein using an example set of columns, which are depicted in FIG. 7A. FIG. 7B depicts the partitioning stages. The partitioning stages are performed, at least in part, by components of DMAC 140. FIG. 7C is a diagram depicting a view of DMAC 140, the view highlighting components that participate in partitioning.

[0221] In general, partitioning involves moving columns from main memory to DMS memory 150 (specifically column memory 774), which in effect is an intermediary memory where the columns are staged to be partitioned among scratch pads of core processors. For each row to be partitioned, an identifier is generated that identifies a core processor to which the row is assigned by partitioning. The identifier generated is referred to herein as a core processor identifier (CID). A row is moved to the DMEM of the core processor identified by the row's respective CID.

[0222] Referring to FIG. 7A, it depicts four columns, key column key1, key column key2, pay load column pay3, and pay load column pay4, as stored in main memory, and which together comprise rows 715. Each of these columns may be stored contiguously in main memory, or may
be stored in multiple "chunks" of contiguous memory. A key column is a column having values that are used to generate CIDs. A pay load column is not used to generate CIDs.

In general, to partition rows, Data Movement System 101 partitions the rows by subsets of the rows. For example, if rows 715 comprise 1&rows (1024 rows), then 7 subsets comprising 256 rows each are partitioned together by Data Movement System 101.

PARTITIONING STAGES

FIG. 7B depicts partitioning stages according to an embodiment of the present invention. Referring to 7B, in the first partitioning stage 721, which is referred to herein as the "DMS load stage", a subset of rows are loaded in the DMS memory 150 from main memory. The rows to load should include at least one key column.

In the second partitioning stage 722, referred to herein as the "CID generation stage", a list of CIDs are generated based on the one or more key columns loaded in the DMS load stage. A CID is generated for and associated with each row of the subset that resides in the DMS load stage.

In the third partitioning stage 723, referred to herein as the "Core partitioning stage", the subset of rows is distributed among the core processors. Each row is moved to the core processor identified by the row's respective CID.

PARTITIONING DESCRIPTORS

The Data Movement System 101 is configured to partition rows through the use of linked descriptors, referred to herein as a partitioning chain. Within a partitioning chain, a different set of linked descriptors are used for each partitioning stage, each set being referred to herein as a partitioning sub-chain. Descriptors are referred to herein as being linked, in a chain, or sub-chain, when the descriptors are linked by Data Movement System 101 (e.g. by a DMAD) by setting Link Addr field to refer to another descriptor that is the same or another chain or sub-chain. The rows are partitioned using components of Data Movement System 101 depicted in Fig. 7C. Partitioning of rows by Data Movement System 101 is illustrated in the context of rows 715. An illustrative partitioning chain 730 that may be used for partitioning is depicted in FIG. 7D.

In FIG. 7D, partitioning chain 730 may be generated by any of the one or more core processors within Data Movement System 101. The partitioning chain, once generated by a core processor, is forwarded to DMAC 140 via a DMAD for processing in the way previously
described for descriptors. For purposes of illustration, core processor 104g is generating partitioning chain 730, and core processor 104g uses DMAD 115g to forward descriptors of partitioning chain 730 to DMAC 140 in the order depicted in FIG. 7D for execution.

[0229] Referring to FIG. 7D, DMS load sub-chain 751 comprises data descriptor 731, data descriptor 732, data descriptor 733, and data descriptor 734. The data descriptors in DMS load sub-chain 751 cause execution of the DMS load stage. Each of these data descriptors specify a source address in main memory for a different column for a subset of rows 715 and a different destination address within column memory 774, an area of memory within DMS memory 150. Each data descriptor also specifies 256 as the number of rows to move to DMS memory 150. Data descriptor 731 is for column key1, data descriptor 732 is for column key2, data descriptor 733 is for column pay3, and data descriptor 734 is for column pay4.

[0230] Data descriptor 731 includes an attribute, referred to herein a key column tag, that identifies column key1 as a key column. Data descriptor 732 includes a key tag to identify column key2 as a key column.

[0231] When a data descriptor with a key flag is forwarded to DMAC 140, it is initially processed by read descriptor parser logic block 121b. Read descriptor parser logic block 121b notifies HARE engine 773 of receipt of a data descriptor with a key flag. HARE engine 773 is a group of electronic circuits that generates hash values and/or CIDS based on the one or more columns indicated by key column register 771. Key column register 771 is a FIFO register. When HARE engine 773 is notified of data descriptor 731, HARE engine 773 adds, as specified by data descriptor 731, the address that holds column key1 in column memory 774 and the width of column key1 to key column register 771. An entry in key column register 771 is added for data descriptor 732 in the same way.

HARE DESCRIPTOR

[0232] CID generation sub-chain 752 comprises one descriptor, HARE descriptor 735. HARE descriptor 735 specifies to generate CID's based on key columns identified by key column register 771. HARE descriptor 735 includes various fields, each specifying an aspect of generating a CID. A HARE descriptor is forwarded by the read descriptor parser logic block 121b to HARE engine 773, which generates the CID's accordingly.

[0233] FIG. 7E shows CID array 717. HARE engine 773 generates CID array 717 when executing HARE descriptor 735. Each element in CID array 717 corresponds to a row in rows
715, and, upon completion of executing HARE descriptor 735, holds a CID for every row in 715. CID 717 is stored in CID memory 776.

[0234] HARE descriptor 735 includes a field that specifies an algorithm for generating CID's. According to an embodiment, one of three algorithms may be specified, which are Radix, Radix/Hash, and Range.

[0235] Under the RADIX algorithm, the value represented by a contiguous range of bit positions in a key column is used as, in effect, a CID. RADIX register 777 specifies the range of bit positions, and key column register 771 specifies the key column. For example, to identify 32 core processors, RADIX register 777 stores a value specifying bit position range 0 through 7. For a column value of a row in a key column in column memory 774, HARE engine 773 sets the corresponding element in CID array 717 to the value of bits 0 through 7 of the column value.

[0236] Under RADIX/HASH algorithm, CID's are generated by, in effect, applying the RADIX algorithm to hash values generated from one or more key columns. Specifically, a hash value column containing the hash values is generated using one or more key columns identified by key column register 771. Hash value column 716 is used to generate a CID array. Hash value column 716 is stored in hash column memory 775, at an address specified by a field of a HARE descriptor. The hash column contains a hash value for each row in the one or more key columns. A bit range of the hash column is used to generate the CID's for a CID array, the bit range being specified by RADIX register 777.

[0237] For example, a field in HARE descriptor 735 specifies the RADIX/HASH algorithm for generating CID's and RADIX register 777 specifies bit position 0 - 4. From the first row of columns key1 and key2, HARE engine 773 generates a hash value and stores the hash value as the first row in hash column 716. Hash column 716 is stored in hash column memory 775. The first five bits of this hash value are stored as the value in the first row in CID array 717. From the second row of key1 and key2, HARE engine 773 generates a hash value and stores the hash value as the second row in hash column 716. The first five bits of this hash value are stored as the value in the first element in CID array 717.

RANGE ALGORITHM

[0238] Under range partitioning, a CID for a row is generated by comparing a column value in a row of a key column to a configurable number of incrementing range values. If the configurable number of range values is "R," the comparsion results in each row being placed into
one of R ranges. The CID for a row is then determined by using the CID assigned to the range by a "range-to-CID mapping" stored in one or more range configuration registers. According to an embodiment, range configuration registers are in DMAC 140 and comprise a range configuration register for each CID, where the CID represents a core processor. Each range configuration register is associated with a range and stores a CID mapped to that range. A configuration register is programmed using a program descriptor. In effect, each range configuration registration holds an entry in the range-to-CID mapping, mapping that range to a CID.

The range-to-CID mapping that can be programmed into range configuration registers is flexible. Multiple ranges may be mapped to the same CID or multiple CIDs can mapped to the same range. Not every CID need be mapped.

When a range is mapped to multiple CIDs, whenever HARE engine 773 determines a range based on a column value in a key column, a CID is assigned in a linear incrementing fashion, starting at the entry that maps that range to a CID and incrementing the CID until it is one less than the CID assigned to the next range. This technique can be used to help reduce CID skew when it is known that one range will have more matches than other ranges. Instead of a single CID being used over and over when there is a match to that range, multiple CIDs are used, and they are used such that the distribution to those CIDs is even.

**DESCRIPTORS USED FOR PARTITIONING**

Once a CID array is generated for a HARE engine descriptor, after the CID generating stage ends for a set of rows, the core partitioning stage may commence. Core partitioning descriptors are used to configure Data Movement System 101 for the core partitioning stage. FIG 7D depicts core partitioning sub-chain 753, which comprises core partitioning descriptors. Core partitioning sub-chain 753 includes core partitioning descriptor 741, core partitioning descriptor 742, core partitioning descriptor 743, core partitioning descriptor 744, and core partitioning descriptor 745. Each of descriptor 741, core partitioning descriptor 742, core partitioning descriptor 743, core partitioning descriptor 744, and core partitioning descriptor 745 is a data descriptor for partitioning a respective column from DMS memory 150 to DMEM memory, the respective column being partitioned among the core processors according to CID's in CID array 717. Each includes a partition flag attribute specifying that a respective column is to be partitioned according to CID array 717. Hence, core
partitioning descriptors 741, 742, 743, 744, and 745 are referred to herein as core partitioning descriptors.

Each of these partitioning descriptors identifies a column to be partitioned among core processors. The Source Addr identifies the column by referencing the column's address in column memory 774. Core partitioning descriptor 741 identifies column key1, core partitioning descriptor 742 identifies column key2, core partitioning descriptor 743 identifies column pay3, and core partitioning descriptor 744 identifies column pay4. Each of the descriptors also specifies the respective column's width. The core partitioning descriptor 745 has an attribute set to indicate that core partitioning descriptor 745 is the last partitioning descriptor in partitioning sub-chain 753.

Core partitioning descriptor 745 identifies hash column 716. The descriptor, when executed, causes the partitioning of hash column 716 among the core processors of Data Movement System 101. In effect, a column comprising hash values generated from other columns of rows 715 is added to rows 715.

Each partitioning descriptor specifies a destination address (i.e. in the Destination Addr.) in DMEM. The destination address is an area in DMEM memory referred to herein as a column FIFO buffer. For each core partitioning descriptor, the core processors in Data Movement System 101 have a respective column FIFO buffer in scratchpad memory at the destination address.

Partitioning engine 772 transmits rows partitioned to a core processor along a "partition data path" connected to partitioning engine 772 and the respective DMEM interface block of the core processor. The partition data path comprises digital circuitry in a data movement engine and the FIFO register of the respective DMAD of the core processor. The respective DMAD block of the core processor receives the rows of rows 715 that are partitioned to that core processor and, via the respective DMEM interface block of the core processor, puts the rows in that core processor's respective column FIFO buffer. For example, partitioning engine 772 transmits rows partitioned to core processor 103a along the partition data path in data movement engine 130a and FIFO register 114b. DMAD 106a receives the rows and puts, via DMEM interface block 107a, the rows in the respective column FIFO buffer of core processor 103a. In this way, core processor 103a receives the rows partitioned to core processor 103a by partition engine 772.
When a DMAD block of a core processor forwards a core partitioning descriptor to DMAC 140, read descriptor parser logic block 121b forwards the core partitioning descriptor to partition engine 772. Partition engine 772 then partitions the column identified by the partitioning descriptor according to the CID array 717.

For example, assume the first four elements in CID array 717 contain the following CID's: 0, 6, 16, 0, which identify core processors 103a, 103g, 104a, and 103a, respectively. To process core partitioning descriptor 741, partition engine 772 reads the first row of column keyl and the first CID value from the first element in CID array 717 and forwards the first row of column keyl to core processor 103a, which is identified by first CID value 0. The core receives the partitioned data, placing the row in the column FIFO buffer of 102a at the address specified by the field Dest Addr of core partitioning descriptor 741. Partition engine 772 forwards the second row of column keyl to core processor 103g, as identified by CID value 6. The core receives the partitioned data, placing the row in the column FIFO buffer of 102g identified by the field Dest Addr of core partitioning descriptor 741. The third row is processed in similar fashion, except it is added to the column FIFO buffer of core processor 104a.

Partition engine 772 forwards the fourth row of column keyl to core processor 103a, as identified by CID value 0 in the fourth element of CID array 717. The core receives the partitioned data, adding that row to the column FIFO buffer of 102a, which already holds the first row from column keyl.

Partition engine 772 processes core partitioning descriptors 742, 743, 744, and 745 in similar fashion. Note, however, rows for these descriptors are placed in a column FIFO buffer that is different between each partitioning descriptor of core partitioning sub-chain 753. For core partitioning descriptor 745, the rows of hash values are obtained from hash column memory 775, and specifically, from hash column 716.

**SUB-BUFFERING**

In order for a core processor to process rows partitioned to that core processor for a core partitioning sub-chain, that core processor must be able to determine when partitioned rows are stored in the respective column FIFO buffers of the core partitioning sub-chain.

According to an embodiment, a core processor is notified when a batch of one or more rows have been completely added to the column FIFO buffers. The batch size (i.e. number of rows) is configurable, and may be set to one. The batch size may be configured by setting a
batch size register using a program descriptor. A batch size register resides in each DMAD block. A batch size register of a DMAD block may also be configured by the respective core processor of the DMAD block by writing directly to the batch size register via a configuration interface.

[0252] Each of the column FIFO buffers is treated as a circular buffer. For a particular core processor, the respective column FIFO buffers are, in effect, associated with the same tail index and head index. The tail index specifies the index of a first unread row (in FIFO order) in any of the respective column FIFO buffers of a core processor. A head index specifies the index of where a row should be added to any of the column FIFO buffers.

[0253] As shall be described in greater detail, the electronic circuitry of DMS 101 manages aspects of flow control with respect to a column FIFO buffer, which includes maintaining a tail index and head index on a per core processor basis, and preventing "buffer overflow", that is, preventing the overwriting of unread rows in the column FIFO buffers of any core processor. Such flow control may include ceasing the distribution of partitioned rows to the column FIFO buffers to prevent buffer overflow.

[0254] For the particular set of rows being partitioned for a core partitioning sub-chain for a core processor, the tail index and head index is updated in response to adding rows to the "last" column FIFO buffer for the partitioning descriptor that is marked as a last descriptor in a core partitioning sub-chain. The tail index and head index is used for all column FIFO buffers for the core partitioning sub-chain, and are not changed until the entire row is added to all these column FIFO buffers. A row is entirely added when the row is added to the last column FIFO buffer.

[0255] Notifying a core processor of the addition of a row to the respective column FIFO buffer (or the update of a tail index) entails a certain amount of overhead. The overhead may be reduced by notifying a core processor when a batch of multiple rows is added. As mentioned before, batch size is configurable. A core processor is notified when a number of multiple rows is added to the respective column FIFO buffers, where that number is equal to the configured batch size.

SUB-BUFFERS

[0256] When the batch size is greater than one, a column FIFO buffer is effectively divided into sub-buffers. When a number of rows equal to the batch size is added to a last column FIFO buffer and the core processor is notified, a sub-buffer comprising that number of rows is made
available to a core processor for processing. The batch size is hence forth referred to herein as the sub-buffer size.

[0257] FIG. 8A depicts an illustrative column FIFO buffer 802a that holds rows partitioned to core processor 103a for core partitioning descriptor 745 and that reside in DMEM unit 102a. Column FIFO buffer 802a comprises 256 rows. The sub-buffer size of column FIFO buffer 802a is 64. Hence, column FIFO buffer 802a comprises four sub-buffers: sub-buffer 811, sub-buffer 812, sub-buffer 813, and sub-buffer 814.

[0258] The column FIFO buffers in DMEM unit 102a for the core partitioning descriptors 741, 742, 743, and 744 also have sub-buffers of the same size.

[0259] When sub-buffer 811 of column FIFO buffer 802a is filled, and it and the respective other sub-buffers of the other column FIFO buffers are made available for processing to a core processor, the core processor is notified and provided a tail index. The tail index points to the first row in the sub-buffer, and is hence forth referred to herein as the sub-buffer index. The sub-buffer index points to a set of rows in a column FIFO buffer that has not been processed by the core processor. Given a sub-buffer size, the core processor processes, for each column FIFO sub-buffer, that number of rows beginning with the row pointed to by the sub-buffer index.

[0260] An index, such as a sub-buffer index, tail index, and head index, refers to an ordinal position of row within a column FIFO buffer. In order to use the index to access a row in any particular column FIFO buffer, the index is resolved to a memory address. A core processor, pursuant execution of software, calculates a memory address for the row using the base memory address of the column FIFO buffer (as specified in the Destination Addr field of the respective core partitioning descriptor), the width of the column, according to the following formula, which assumes that the index value for the first row is zero.

\[ \text{Row Memory Address} = \text{base memory address} + (\text{index} \times \text{column width}) \]

Thus, in this way, an index may be used to identify a row (or the beginning of a set of rows) to access in each of the column FIFO buffers.

ROW PROCESSING AND FLOW CONTROL

[0261] According to an embodiment of the present invention, the electronic circuitry of a DMEM interface block is configured for handling aspects of flow control for each column FIFO buffer. Such aspects of flow control include: (1) maintaining a tail index and head index for each column FIFO buffer of a partitioning descriptor that is flagged as the last partitioning descriptor,
(2) notifying a core processor when a sub-buffer has been filled with rows, and (3) signaling to partition engine 772 to stop partitioning and distributing rows to prevent column FIFO buffer overflow.

[0262] A core processor executing software also participates in handling aspects of the flow control of a column FIFO buffer. These include signaling to the respective DMEM interface block that a sub-buffer has fully been read, processed, and/or is otherwise available to receive new partitioned rows.

[0263] FIG. 8B is a diagram depicting operations performed by a core processor 103a and DMEM interface block 107a to process rows forwarded to a core processor 103a by partition engine 772.

[0264] Referring to FIG. 8B, at 820, core processor 103a receives notification from DMAD DMEM interface block 107a that a sub-buffer of the last column FIFO buffer has been filled, along with a sub-buffer index. At 822, core processor 103a processes rows in the sub-buffers identified by the sub-buffer index, which not only include the last column FIFO buffer for core partitioning descriptor 745, but the other column FIFO buffers for core partitioning descriptors 741, 742, 743, and 744. At 823, core processor 103a sends notification that the sub-buffers have been fully processed.

[0265] At 824, core processor 103a waits for the next notification that a sub-buffer is available. While waiting, or in lieu of waiting, core processor can perform other operations and work.

[0266] DMEM interface block 107a performs operations 830 - 838. At 830, DMEM interface block 107a receives rows from partition engine 772 partitioned to core processor 103a. A column of rows are received for each partitioning descriptor in core partitioning sub-chain 753; DMEM interface block 107a fills the respective column FIFO buffer with the rows. The last column of rows received for core partitioning sub-chain 753 are those for last core partitioning descriptor 745.

[0267] At 832, after adding a quantity of the last column of rows that is equal to or greater than the sub-buffer size of the respective column FIFO buffer, DMAD DMEM interface block 107a sends a notification to core processor 103a that a sub-buffer has been filled, along with the sub-buffer index.
DMEM interface block 107a maintains the sub-buffer index and head index. This maintenance includes performing any wrapping around operation as is needed for a circular buffer.

With respect to the head index, DMEM interface block 107a increments the head index as each row is added to the column FIFO buffer for the first partitioning descriptor in core partitioning sub-chain 753. The value of the head index controls, in effect, whether to stop adding new rows to the column FIFO buffer to prevent buffer overflow. The value of the head index is prevented from reaching that of the sub-buffer index.

At 834, DMEM interface block 107a detects whether the difference between the head index and sub-buffer index satisfies "Full Criteria". The full criteria is based on a threshold difference between the head index and sub-buffer index. Once the difference is at or less then the threshold, then full criteria is satisfied. Once it is determined that the full criteria is satisfied, at 836, DMEM interface block 107a signals partition engine 772 to stop partitioning. Specifically, DMEM interface block 107a back pressures the respective partition data path and when the partition data path becomes full, partition engine 772 stops partitioning. The threshold difference upon which the full criteria is based is configurable by a DMAD register using a program descriptor or by the respective core processor of a given DMAD writing the register through a configuration interface.

At 838, the sub-buffer index is incremented by the DMEM interface block 107a in response to a receipt of a notification by the core processor 103a that it has processed the rows of a sub-buffer. If the DMEM interface block 107a had detected that the full criteria had been satisfied, it re-evaluates the criteria after the sub-buffer index is incremented. When the DMEM interface block 107a detects that full criteria is no longer satisfied, DMEM interface block 107a signals partition engine 772 to begin partitioning the first core partitioning descriptor 741.

**PARTITION PIPELINING**

According to an embodiment of the present invention, pipelining allows various resources, such as partition engine 772, and HARE engine 773, to be concurrently used to process a subset of rows for different partitioning sub-chains. Each of the three partitioning stages can be executed concurrently (i.e. within the same clock cycles) to process different subsets of rows.
[0273] FIG. 9A is a diagram depicting partitioning pipelining according to an embodiment of the present invention. Referring to FIG. 9A, pipelined sets 901, which comprise three pipelined sets of partitioning descriptors, each set processing the same rows from main memory. Two of the sets are shown twice in FIG. 9A as described below. Each pipelined set includes a partitioning sub-chain for each partitioning stage; each partitioning sub-chain being unlinked with another partitioning sub-chain in the respective pipelined set, i.e., the Link address field of the last descriptor in partition chain is not linked to the first descriptor of another partitioning sub-chain in the respective pipelined set. Order of execution between partition sub-chains in a pipelined set is controlled through wait for conditions, as explained below. The pipelined sets include:

a. DMS load sub-chain A1, CID generation sub-chain A2, and core partitioning sub-chain A3 for partitioning a respective set of rows. In FIG. 9A this set is shown twice - the second time this set executes on a different respective set of rows;

b. DMS load sub-chain B1, CID generation sub-chain B2, and core partitioning sub-chain B3 for partitioning another respective set of rows. In FIG. 9A this set is shown twice - the second time this set executes on a different respective set of rows, and

c. DMS load sub-chain C1, CID generation sub-chain C2, and core partitioning sub-chain C3 for partitioning yet another respective set of rows.

[0274] Intervals II through 17 are an ordered sequence of periods of time. In each of the intervals II through 17, Data Movement System 101 may be concurrently executing up to three partitioning sub-chains, one for each partitioning stage.

[0275] For each pipelined set, partitioning sub-chains are executed in partition stage order, and awaits for completion of the partitioning sub-chain that completed beforehand. For example, DMS load sub-chain A1 is executed in interval II. CID generation sub-chain A2, which must wait for completion of execution of DMS load sub-chain A1, is executed in interval 12. Core partitioning sub-chain A3, which must wait for completion of execution of CID generation sub-chain A2, is executed in interval 13.

[0276] Through orchestration of wait for conditions, the partitioning sub-chains of a pipelined set are executed, in effect, in a loop. Thus, before DMS load sub-chain A1 is executed
in interval 14 to process a different set of rows, execution of DMS load sub-chain A1 must wait for completion of core partitioning sub-chain A3 in interval 13.

[0277] Interval II and I2 comprise the initial phase of pipelining, referred to as the fill stage. In the fill stage, a partitioning sub-chain is not being executed for each partitioning stage. Because only one partitioning sub-chain per partitioning stage may be executed in a single interval, and the first partitioning stage for a subset of rows begins with DMS load stage, interval I1 includes only the execution of one DMS load sub-chain, which is DMS load sub-chain A1. In interval I2, two partitioning sub-chains are executed, which are DMS load sub-chain B1 and CID generation sub-chain A2.

[0278] Intervals I3 through I5 comprise the full phase of pipelining, where three partitioning sub-chains may be executed concurrently, one for each of the three partitioning stages. In interval I3, DMS load sub-chain CI, CID generation sub-chain B2, and core partitioning sub-chain A3 are executed concurrently. In interval I4, CID generation sub-chain C2, core partitioning sub-chain B3, and DMS load sub-chain A1 are executed concurrently.

CONSTRUCTING AND SUBMITTING PARTITIONING SUB-CHAINS

[0279] According to an embodiment of the present invention, for each partitioning stage, a separate core processor forms and submits a chain of partitioning sub-chains.

[0280] Referring to FIG. 9B, it depicts DMS load chain 911, comprising the partitioning sub-chains for the DMS load stage from each of the pipelined sets, and in particular, comprising DMS load sub-chain Al, DMS load sub-chain B1, and DMS load sub-chain CI. Core processor 103a forms these chain descriptors within DMEM unit 102a and submits the chain descriptors to one of the descriptor channel blocks of DMAD 106a. The chain of descriptors also includes a loop descriptor for looping execution of DMS load chain 911. The loop descriptor and DMS load chain 911 are configured for looping as described above.

[0281] CID generation chain 912 comprises the partitioning sub-chains for the CID generation stage, which comprise CID generation sub-chain A2, CID generation sub-chain B2, and CID generation sub-chain C2. Core processor 103g forms CID generation chain 912 within DMEM unit 102g and submits CID generation chain 912 to a data channel DMAD 106g. The chain of descriptors also includes a loop descriptor for looping execution of CID generation chain 912. The loop descriptor and CID generation chain 912 are configured for looping as described earlier.
[0282] Core partitioning chain 913 comprises the partitioning sub-chains for the core partitioning stage, which comprise core partitioning sub-chain A3, core partitioning sub-chain B3, and core partitioning sub-chain C3. Core processor 104a forms core partitioning chain 913 within DMEM unit 105a and submits core partitioning chain 913 to one of the descriptor channel blocks of DMAD 115a. The chain of descriptors also includes a loop descriptor for looping execution of core partitioning chain 913. The loop descriptor and core partitioning chain 913 are configured for looping as described earlier.

[0283] For a set of rows processed by an execution of a set of partitioning sub-chains, the partitioning sub-chain must be executed in partitioning stage order. When the partitioning sub-chains are submitted by the same core processor, the partitioning sub-chains are executed in the order submitted to the given descriptor channel of the respective DMAD. Thus, as long as the partitioning sub-chains are submitted in partition stage order, the sub-chains are executed in the partition stage order. It is important to keep certain descriptors from starting until certain other descriptors have completed. For example, the CID generation sub-chain A2 is prevented from starting until the DMS load sub-chain A1 has completed.

[0284] However, for pipelined sets 901, the partitioning sub-chains of each partitioning stage are submitted by different core processors. Therefore, for a given set of rows processed by a set of partition sub-chains, execution of the sub-chains must be synchronized such that the set of partition sub-chains are executed in partition stage order.

[0285] According to an embodiment, such synchronization is orchestrated through wait-events, as illustrated in FIG. 9B. Specifically, each partition sub-chain is associated with a wait condition that must be satisfied before execution of the partitioning sub-chain begins. Execution of the partition sub-chain is blocked until the wait condition is satisfied. The wait condition is based on an event. The wait condition for a partition sub-chain is specified by the "wait for" field of the first descriptor in the partition sub-chain.

[0286] For example, for DMS load sub-chain A1 the wait condition is Event0 equal to 0, for CID generation sub-chain A2 the wait condition is Event3 equal to 0, and for core partitioning sub-chain A3 the wait condition is Event6 equal to 0.

[0287] Completing execution of a partition sub-chain causes: (a) setting the event to a state that causes the wait condition for the partition sub-chain to be unsatisfied, thereby blocking the partition sub-chain from executing, and (b) the setting of another event to a state that satisfies a
wait condition of a subsequent partition sub-chain in the same pipelined set, thereby unblocking
the subsequent partition sub-chain from executing.

[0288] Completing execution of a partition sub-chain may entail setting two events for
purpose of synchronization. In an embodiment, a descriptor may only set one event. Therefore, a
partition sub-chain may include an additional descriptor, the purpose of which is to set an event.

[0289] For example, initially, core processor 103a sets events such that only execution of
DMS load sub-chain A1 is permitted and execution of CID generation sub-chain A2 and core
partitioning sub-chain A3 is blocked. Accordingly, core processor 103a clears EventO, i.e. sets to
0, and sets both Event3 and Event6 to 1. Completing execution of DMS load sub-chain A1 sets
EventO to 1, thereby blocking DMS load sub-chain A1 from executing again, and clears event3,
thereby unblocking CID generation sub-chain A2 from executing. Completion of execution CID
generation sub-chain A2 sets event3 to 1, thereby blocking CID generation sub-chain A2 from
executing again, and clears Event6, thereby unblocking core partitioning sub-chain A3 from
executing. Completion of execution core partitioning sub-chain A3 sets Event6 to 1, thereby
blocking core partitioning sub-chain A3 from executing again, and clears EventO, thereby
unblocking subsequent DMS load sub-chain A1 from executing.

ROW IDENTIFICATION NUMBERS

[0290] According to an embodiment, Data Movement System 101 may be configured to
generate a column of RIDs that may be used to perform row resolution between source columns
and resultant columns generated from the source column, or other columns that are row aligned
with the source column.

[0291] FIG. 10 illustrates RIDs and how RIDs may be used to perform row resolution.
Referring to FIG. 10, it depicts source column SC7 1002, which is partitioned by Data
Movement System 101 into three resultant columns, resultant column RC1 1031, resultant
column RC2 1032, and resultant column RC3 1033.

[0292] FIG. 10 also depicts non-partitioned columns NP3 1003, NP4 1004, and NP5 1005.
These columns are row-aligned with source column SC7. However, the columns are not
partitioned in the current illustration.

[0293] RID column RID7 is a column comprising RIDs. The RIDs in a RID column are an
ordered sequence of numbers when the RID column is initially generated according to a
descriptor. In an ordered sequence of numbers, each number differs from an adjacent number in
the sequence by the same constant, referred to herein as a counter value. A counter value is often the value one. The first RID in the sequence is referred to as the starting value.

[0294] To use RIDs in RID column RID7 to perform row resolution, RID column RID7 is assumed to be row aligned with source column SC7. Accordingly, row 1014 contains the RID 1304 and the value "E" in source column SC7.

[0295] Based on the starting value of an ordered sequence of RIDs in a RID column and the respective counter value, the RID of a row in the RID column may be used to perform row resolution for that row on other columns that are row aligned to the RID column.

[0296] For example, given a starting value of 1300 and counter value of 1, RID 1308 may be resolved to row 1018.

ROW RESOLUTION FOR DATA MANIPULATION OPERATIONS THAT PRESERVE ROW ALIGNMENT

[0297] A data manipulation operation may be performed on multiple source columns such that row alignment is preserved between respective resultant columns. The descriptor based partitioning described earlier is an example of such a tabular data operation that preserves row alignment between resultant columns. When a source column is a RID column that contains an ordered sequence of RIDs and is row aligned with another particular source column, and when row alignment between resultant columns is preserved by a data manipulation operation that is applied to both source columns, a resultant RID column may be used to perform row resolution between a resultant column generated for the other particular source column and other particular source column.

[0298] Referring to FIG. 10, RID column RID7 and source column SC7 are partitioned such that the same row belongs to the same partition. Thus, the respective pair of resultant columns for each partition are row aligned (each partition is stored in DMEM of a different core processor). Resultant RID column RRID1 1021 and resultant column RC1 1031 belong to the same partition and are row aligned, resultant RID column RRID2 1022 and resultant column RC2 1032 belong to the same partition and are row aligned, resultant RID column RRID3 1023 and resultant column RC3 1033 belong to the same partition and are row aligned.

[0299] To perform row resolution between a resultant column and a respective source column using a respective resultant RID column of the resultant column, row-alignment-based resolution is used to obtain a RID for a row from the resultant RID column, and the RID is used
to perform RID-based row resolution on the source column. For example, to perform row resolution between source column SC7 and resultant column RC3 for row 1018 in resultant column RC3, row-alignment- resolution is used to obtain the RID for the row. Row 1018 is the third element in resultant column RC3. Therefore, the third element in resultant RID column RRID3 contains the RID for row 1018, which is 1308. Based on a RID value of 1308, the starting value of 1300, and the counter value of 1, RID-based resolution yields that row 1018 is the ninth element in source column SC7.

RID-based resolution using resultant RID columns RRID1, RRID2, or RRID3 may be used to perform row resolution not only between source column SC7 and resultant columns RC1, RC2, or RC3, but also with other columns row aligned with source column SC7. Thus, RID-based resolution using resultant RID columns RRID1, RRID2, or RRID3 may be used to perform row resolution between resultant columns RC1, RC2, and RC3, respectively, and any of non-partitioned columns NP3, NP4, and NP5.

ROW IDENTIFICATION NUMBERS GENERATION

As mentioned previously, Data Movement System 101 generates RIDs within various memories of Data Movement System 101. The RIDs are generated by a dedicated RID engine in each data movement engine (see FIG. 7C), each RID engine comprising a set of electronic circuits that are designed to generate a column of RIDs in response to reading a descriptor.

Referring to FIG. 7C, each data movement engine includes a RID engine and a RID memory unit. A RID memory unit is a type of DMS memory used to store RIDs, although it is not limited to storing only RIDs. Data movement engine 130a includes RID engine 703a and RID memory unit 704a, data movement engine 130b includes RID engine 703b and RID memory unit 704b, data movement engine 130c includes RID engine 703c and RID memory unit 704c, data movement engine 130d includes RID engine 703d and RID memory unit 704d.

According to an embodiment, a column of an ordered sequence of RIDs is generated in response to a data descriptor that specifies various aspects of generating a column of RIDs. A data descriptor that is for generating a column of RIDs includes an attribute referred to herein as a "RID flag", which specifies to generate a column of an ordered sequence of RIDS at a destination address specified in the destination address field. The destination address may be within the DMEM of a particular core processor, DMS memory 150, or RID memory. A data descriptor that specifies to generate RIDs in this way is referred to herein as a RID descriptor.
Unlike for data movement that is performed for data descriptors previously described, generation of RIDs by Data Movement System 101 does not involve moving data from the source address. Thus, for a RID descriptor, the source address field of a data descriptor is not treated as a source address from which to obtain data to move. Rather, the source address field is treated as counter value for generating a sequence of RIDs, which is typically one. Thus, when the source address field value is one, successive RIDs in the sequence differ by one. If the source address field value is two, successive RIDs in the sequence differ by two.

A RID column may have a single-byte or multi-byte column width. The Column Width field in a RID descriptor specifies a column width.

In an embodiment, a RID starting value from which to start generating an ordered sequence of RIDs in a RID column is specified in an auxiliary data descriptor that precedes a RID descriptor. The RID descriptor includes a "RID start flag" to specify that the auxiliary descriptor sets a RID starting value. A "RID Starting Value" field in the auxiliary data descriptor specifies a RID starting value. Alternatively, the RID starting value may be specified by setting a register using a program descriptor or by using a field in a RID descriptor. Specifying the RID starting value in an auxiliary descriptor may be advantageous for accommodating larger RID starting values for larger column widths. There may be insufficient space available in a RID descriptor for a field large enough to specify the larger staring values.

EXEMPLARY PARTITIONING CHAIN WITH RID GENERATION

As mentioned previously, RID generating is particularly advantageous for identifying rows after the rows have been partitioned between core processors. During partitioning, a RID column may be generated in column memory for columns being partitioned, which, in effect, adds a RID column for the rows in the columns being partitioned. When the RID is partitioned to DMEM of a core processor, the row will include a RID column.

FIG. 11A shows partitioning chain 1130 comprising descriptors that may be used to cause partitioning of the rows that include a RID column. FIG. 11B shows columns generated and/or otherwise processed while partitioning rows according to partitioning chain 1130.

Partitioning chain 1130 partitions columns among core processors of Data Movement System 101, the columns include a RID column. Partitioning chain 1130 includes DMS load sub-chain 1151 for the load stage, CID generation sub-chain 1152 for the CID generation stage, and core partitioning sub-chain 1153 for the core partitioning stage.
DMS load sub-chain 1151 comprises data descriptor 1131, data descriptor 1132, auxiliary descriptor 1133, and RID descriptor 1134. Each of data descriptor 1131 and data descriptor 1132 specifies a source address in main memory for a different column for a subset of rows 715 and a different destination address within column memory 774. Data descriptor 1131 is for key column KEY8 1111, data descriptor 1132 is for pay column PAY8 1112. Data descriptor 1131 includes a key column tag. Each data descriptor also specifies 256 as the number of rows to move to DMS memory 150.

Auxiliary descriptor 1133 specifies a RID starting value of 1300 in the RID starting value field. When auxiliary descriptor 1133 is forwarded to DMAC 140, it is initially processed by read descriptor parser logic block 121b. Read descriptor parser logic block 121b detects the auxiliary descriptor contains a RID starting value, causing read descriptor parser logic block 121b to update an internal parser register with the starting value. For purposes of illustration, the RID starting value is 1300. In an embodiment, a RID descriptor is immediately preceded by an Aux descriptor that contains a RID starting value.

RID descriptor 1134 is a RID descriptor. RID descriptor 1134 includes a RID flag. RID descriptor 1134 specifies a column width field of 1, a destination address within column memory 774, and 256 as the number of rows to generate in a RID column. The source address field is set to 1, specifying a counter value of 1.

When RID descriptor 1134 is forwarded to DMAC 140, it is initially processed by read descriptor parser logic block 121b. Read descriptor parser logic block 121b detects the RID flag, causing read descriptor parser logic block 121b to notify a RID engine in one of the data movement blocks (130a, 130b, 130c, or 130d) of receipt of a RID descriptor 1134.

When the notified RID Engine receives the notification, the RID Engine generates RID column RID8 1113 accordingly. Thus, RID column RID8 has a column width of two bytes, which includes 256 rows or elements. The first RID in RID column RID8 is 1300, the RID starting value specified in auxiliary descriptor 1133. Successive RIDs in RID column RID8 are created by incrementing the RID starting value by 1, the specified counter value. The next two successive RIDs in RID column RID8 are thus 1301 and 1302, respectively.

CID generation sub-chain 1152 comprises one descriptor, HARE descriptor 1135. HARE engine 773 generates CID array CID8 1118 when executing HARE descriptor 1135.
Core partitioning sub-chain 1153 specifies how to perform the core partitioning for key column KEY8, payload column PAY8, and RID column RID8. Core partitioning sub-chain 753 includes core partitioning descriptor 1141, core partitioning descriptor 1142, and core partitioning descriptor 1143. Core partitioning descriptor 1141 is for partitioning key column KEY8, and core partitioning descriptor 1142 is for partitioning payload column PAY8, and core partitioning descriptor 1143 is for partitioning RID column RID8.

Each of these partitioning descriptors identifies the respective column to be partitioned among core processors as described before. With respect to core partitioning descriptor 1143, the Source Addr field identifies the RID column RID8 by referencing the column's address in column memory 774.

Each core partitioning descriptor specifies a destination address (i.e. in the Destination Addr. field) in DMEM. For each core partitioning descriptor, the core processors in Data Movement System 101 have a respective column FIFO buffer in scratchpad memory at the destination address, that is, for each core processor, there is a respective column FIFO buffer for each of key column KEY8, payload column PAY8, and RID column RID8. These column FIFO buffers are row aligned.

For example, after processing HARE descriptor 1135, assume the first four elements in CID array CID8 contain the following CID's: 0, 6, 16, 0 in the first four rows, which identify core processors 103a, 103g, 104a, and 103a, respectively. After processing core partitioning sub-chain 1153, the first and fourth rows of column KEY8, payload column PAY8, and RID column RID8 are stored contiguously in respective column FIFO buffers of core processor 103a. In the column FIFO buffer for RID column RID8 in core processor 103a, the first two elements contain the first and fourth RID entries 1300 and 1303, respectively, just as these rows did when stored in column memory 774 before partitioning.

RIDS FOR OTHER TABULAR DATA MANIPULATION OPERATIONS

Partitioning is one example of a tabular data manipulation operation that alters row alignment when generating resultant columns. Another is a gather operation. In a gather operation, Data Movement System 101 filters out rows of a column while the column is in flight from a source memory location to a destination memory location, and compacts the resulting rows of the column, (i.e. the rows that were not filtered out) while storing the resulting rows in the destination memory location such that the resulting rows are stored in consecutive memory...
locations within the destination memory even if the resulting rows were not stored in consecutive memory locations at the source memory location. The rows may be filtered out based on a bit vector.

According to an embodiment, a RID descriptor may specify a data manipulation operation, such as a gather operation. Thus, after performing a data manipulation to a particular column moved to a memory such as DMEM of a core processor, a RID column is in effect manipulated in the same way. The resultant manipulated column and RID column are row aligned allowing the RIDs to be used for RID-based row resolution.

The data movement system may convert each RID within the list of RIDs from its logical RID to a corresponding physical RID prior to performing a gather and/or scatter operation. A logical RID, as described herein, is a unique identifier assigned to each row in a database table. A physical RID, as described herein, is derived from a logical RID. In an embodiment, the physical RID is derived by subtracting a base value from the logical RID. This ability to convert logical RIDs to physical RIDs allows a core processor to work on a smaller subset of rows of a column of tabular data that are stored in contiguous addresses in main memory. Additionally, a core processor may initiate data movement with a gather operation using a subset of rows without first converting the logical RIDs of the rows to their corresponding physical RIDs.

Generation of column RIDs by Data Movement System 101 has many advantages for many different types of database operations. One example of such an operation is a partitioned "filter and projection" database operation. In a partitioned filter and projection operation, rows may be partitioned between core processors so that a portion of the columns of rows are evaluated in parallel against filtering criteria to determine which subset of rows satisfy the filtering criteria. The subset of rows are then further processed. Such further processing include processing "carry" columns, which, in the context of the filtering portion are the partitioned filter and projection operation, are the columns that are not evaluated for the criteria.

One technique for performing a partitioned filter and projection operation is to partition the rows in their entirety and then examine the column pertinent to the filtering. In this technique, the carried columns are distributed to the scratchpad memory of core processors even though many of the "filtered out" rows of the carried columns are never otherwise processed. If
the filtering criteria is selective, then potentially substantial processing bandwidth is expended transferring data that is not pertinent to the partitioned filter and projection operation.

[0325] With RIDs, only a subset of columns pertinent to the filtering criteria need to be partitioned between core processors. The RIDs of rows satisfying criteria may be used to obtain the rows from other columns.

[0326] For the purpose of identifying a subset of rows in a column, RIDs may require less memory to identify a smaller subset than a bit vector. A bit vector (at least one that is uncompressed) occupies the same amount of memory to identify a subset of rows in a set of rows regardless of the number of rows in the subset. When the number of rows in the subset is much smaller than that of the set, the bit vector is sparse, that is, only a small number of bits are set to identify rows. A list of RIDs may occupy less memory in this case than the bit vector.

[0327] A sparse bit vector may be used to generate a list of RIDs that occupies less memory. A RID descriptor may specify to generate a RID column and apply a bit vector in a gather operation, thereby generating a RID column in a destination memory location, the RID column comprising RIDs that identify the rows identified by the bit vector.

DMS MEMORY ORGANIZATION

[0328] According to an embodiment of the present invention, DMS memory comprises four categories of memory, each category being accessible to an engine (or engines) of DMAC 140 to store data read or written by that engine. Each category may comprise one or more units of memory. One category is used to store columns that are to be partitioned and/or serve as input for generating a hash column, another is used to store hash columns, another is used to store RID columns or bit vectors, and finally another is used to store CIDs. These categories of memory and the arrangement thereof with respect to various engines of the DMAC is depicted in FIG. 12A and FIG. 12B.

[0329] Referring to FIG. 12A, it depicts column memory unit 774a, column memory unit 774b, and column memory unit 774c. Each of column memory unit 774a, and column memory unit 774b, and column memory unit 774c are used to store columns that are partitioned and/or serve as input for generating a hash column.

[0330] DDR load engine 1231a, DDR load engine 1231b, DDR load engine 1231c, and DDR load engine 1231d are the DDR data load engines of data movement engine 130a, data movement engine 130b, data movement engine 130c, and data movement engine 130d,
respectively. According to an embodiment, each of DDR load engine 1231a, 1231b, 1231c, and 1231d may move a column from DDR memory to any of column memory units 774a, 774b, and 774c. The column is moved in response to receipt of control information from read descriptor parser logic block 121b. Read descriptor parser logic block 121b dispatches the control information based on a descriptor parsed by read descriptor parser logic block 121b, the descriptor specifying the source address of the column in DDR memory and a destination column memory unit 774a, 774b, or 774c, and the destination address within destination column memory unit 774a, 774b, or 774c. Write access by DDR load engine 1231a, 1231b, 1231c, and 1231d to any of column memory units 774a, 774b, and 774c is arbitrated by arbitration unit 1290a.

[0331] HARE engine 774 accesses (via arbitration unit 1290b) any of column memory units 774a, 774b, or 774c to read one or more key columns from which to generate a hash column. Partition engine 772 accesses (via arbitration unit 1290c) any of column memory units 774a, 774b, or 774c to read one or more columns therein to partition.

[0332] Referring to FIG. 12B, it depicts RID memory units 704a, 704b, 704c, and 704d. Each of RID memory units 704a, 704b, 704c, and 704d comprise one or more memory units that are each used to store RID columns or BVs. The RID columns may be loaded from DMEM to any of RID memory units 704a, 704b, 704c, or 704d. Also, each of the DDR load engines have a RID Engine which can access the local RID memory that resides in that data movement engine. Thus, RID memory units 704a, 704b, 704c, or 704d can be used to store a RID column that is generated according to a RID descriptor (e.g. RID descriptor 1134), which specifies which of RID memory units 704a, 704b, 704c, or 704d to store the RID column. As described earlier, RID columns can also be stored in column memories 774a, 774b, and 774c.

[0333] As mentioned before, data movement engine 130a, 130b, 130c, and 130d each include a DDR load engine, which are DDR load engine 1231a, 1231b, 1231c, and 1231d, respectively. In addition, data movement engine 130a, 130b, 130c, and 130d each include a DMEM load engine, which are DMEM load engine 1241a, 1241b, 1241c, and 1241d, respectively.

[0334] According to an embodiment in which an RID memory unit is internal to a data movement engine, the DDR load engine and DMEM load engine of each data movement engine may access only the respective RID memory unit to read and write a RID column. DDR load
engine 1231a and DMEM load engine 1241a have access to RID memory unit 704a via arbitration unit 1290f, DDR load engine 1231b and DMEM load engine 1241b have access to RID memory unit 704b via arbitration unit 1290g, DDR load engine 1231c and DMEM load engine 1241c have access to RID memory unit 704c via arbitration unit 1290h, DDR load engine 1231d and DMEM load engine 1241d have access to RID memory unit 704d via arbitration unit 1290i.

[0335] A data movement engine can only perform a gather and/or scatter operation using a RID column or BV stored in the RID memory to which the respective DDR load engine and DMEM load engine have access. For example, in order for data movement engine 130a to perform a gather operation using a RID column, the RID column should be stored in RID memory unit 704a.

**DMS-DMS MEMORY MOVEMENT**

[0336] According to an embodiment, a data movement engine performs data movement operations for only the group of core processors connected (via a DMAD) to a particular DMAX. For example, data movement engine 130a performs data movement for core processors 103a and 103g and no other core processor in data movement system 101, such as 104a and 104g.

[0337] Different data movements to different core processors may be performed by different data movement engines but can use the same RID column or BV to perform the data movement. To use the same RID column or BV, the RID column and/or BV is copied to the multiple RID memory units that are accessible to the data movement engines performing the data movement.

[0338] One way to move copies of a RID column or BV to multiple RID memory units is to execute multiple descriptors, each specifying to move the same RID column from DDR memory to a particular RID memory. However, this requires multiple movements from DDR memory to data movement system 101.

[0339] To avoid multiple movements from DDR memory, data movement system 101 is configured to internally move data between various memory units in DMS memory. Data movement performed in this way is referred to herein as internal DMS memory movement. Internal DMS memory movement can be performed more efficiently than data movements between main memory and data movement system 101. Data movement system 101 may be configured to execute internal DMS memory movement by submitting to data movement system 101 a DMS-DMS descriptor. A copy ring is used to perform internal DMS memory movement.
FIG. 13 shows copy ring 1300, which comprises digital electronic circuitry configured for internal DMS memory movement. Copy ring 1300 includes copy ring nodes, each of which are a block of digital electronic circuitry configured to participate in moving data to and from memory units within DMS memory and other copy ring nodes. According to an embodiment, there are several kinds of copy ring nodes: a DMS copy engine node and copy memory interface node. Copy ring 1300 includes DMS copy engine 1311, and copy memory interface nodes 1312, 1313, 1314, 1315, 1316, 1317, 1318, 1319, and 1320, one for each DMS memory unit that serves as a source or destination for internal DMS memory movement.

DMS copy engine 1311 comprises digital electronic circuitry configured to perform various functions that are hereafter described. In general, DMS copy engine 1311 initiates internal DMS memory movement in response to receiving from read descriptor parser logic block 121b control information generated by read descriptor parser logic block 121b for a DMS-DMS descriptor.

Each copy ring node is linked by a separate bus to each of two other copy ring nodes, thereby forming a loop or ring along which data is transmitted between and among copy ring nodes. Each copy ring node receives control information and may receive "copy data" that was retrieved from a DMS memory unit by another copy memory interface node and sent via a bus from another copy ring node. The term copy data refers to data stored in a DMS memory unit that is copied to another DMS memory unit using internal DMS memory movement.

Each of the copy memory interface nodes is coupled to a respective DMS memory unit and is configured to write copy data to the respective DMS memory unit and/or to read copy data from that DMS memory. In another embodiment, a copy memory interface node may be coupled to multiple memory units when such memory units are physically proximate to each other.

According to an embodiment, the loop is directional. That is, a given copy ring node is connected by two separate buses to two other copy ring nodes; one copy ring node ("source node") from which the given copy ring node receives control data and/or copy data, and another copy ring node ("destination node") to which the given copy ring node forwards control data and/or read data.

Referring to FIG. 13, copy memory interface node 1312, 1313, and 1314 are copy memory interface nodes for column memories within DMS memory 150. Copy memory
interface node 1312, 1313, and 1314 are coupled to column memory 774a, 774b, and 774c, respectively. Copy memory interface node 1315 is coupled to hash column memory 775. Copy memory interface node 1316 is coupled to CID memory 776. Copy memory interface nodes 1316, 1317, 1318, and 1319 are coupled to RID memory 704a, 704b, 704c, and 704d, respectively.

[0346] The source and destination node of each copy ring node depicted in FIG. 13 is indicated by directional lines representing a bus between copy rings nodes, with a line directed from a source node of a copy ring node to the copy ring node and a line directed from the copy ring node to the destination node of the copy ring node. For example, the source and destination node for copy memory interface node 1312 is DMS copy engine 1311 and copy memory interface node 1313, respectively.

DMS-DMS DESCRIPTOR

[0347] As with other data movement operations described previously, a descriptor is used to configure Data Movement System 101 to perform an internal DMS memory movement. FIG. 14 depicts DMS-DMS descriptor 1401, which is used to configure Data Movement System 101 to perform an internal DMS memory movement.

[0348] Referring to FIG. 14, it depicts DMS-DMS descriptor 1401. The "Desc Type" field of DMS-DMS descriptor 1401 specifies DMS-DMS descriptor 1401’s descriptor type.

[0349] The "Src Addr" field specifies the source address of copy data. According to an embodiment, the source address should refer to a single memory address space ("DMS memory address space") that covers multiple DMS memory units. A range of the DMS memory address space is exclusively dedicated to a single DMS memory unit, and an address within any DMS memory unit falls within the respective dedicated range. Thus, each memory address of any column memory 774a, 774b, 774c, hash column memory 775, CID memory 776, and RID memory 704a, 704b, 704c, and 704d falls within a particular range, and is absolute within the DMS memory address space.

[0350] "Dest Addr" field specifies the destination address within a DMS memory unit to write copy data. For each DMS memory unit to which copy data is written for a DMS-DMS descriptor, the destination address is the same. Unlike the source address of "Src Addr" field, the destination address is relative to a particular DMS memory unit (e.g. an offset).
"Write Map" fields specifies into which DMS memory to write copy data. For example, Write Map may be a bit map, with each bit corresponding to one of column memory units 774a, 774b, 774c, hash column memory unit 775, CID memory unit 776, and RID memory units 704a, 704b, 704c, and 704d.

"Column Width" indicates the size of the column of the copy data, "Rows" specifies the number of rows of the copy data.

**INTERNAL DMS MEMORY MOVEMENT**

FIG. 15 is a flow chart depicting operations that copy ring 1300 in FIG. 13 performs for internal DMS memory movement. The operations are performed in response to submission of a DMS-DMS descriptor by a core processor to a descriptor channel of the respective DMAD of the core processor.

Referring to FIG. 15, at 1505, DMS copy engine 1311 receives control information from read descriptor parser logic block 121b. The control information includes information specified in the DMS-DMS descriptor, including the source address, the destination address, and the write map, as specified by the "Src Addr", "Dest Addr", and "Write Map" fields, respectively.

Operations 1510 - 1535 represent a loop comprising operations that are performed by each copy ring node. In each iteration of the loop, a successive copy ring node performs the operations in the loop. DMS copy engine 1311 performs the initial iteration, and the operations of the loop are illustrated with DMS copy engines 1311's initial performance of the operations.

At 1510, the current copy ring node, which is DMS copy engine 1311, forwards control information to the destination node of the current copy ring node, copy memory interface node 1312. In a latter iteration of the loop performed by another copy ring node, operation 1510 may also entail receiving copy data. Copy data is received when in a previous iteration, copy data was read from a source DMS memory unit by a copy memory interface node (such as copy memory interface node 1312).

At 1515, the destination node, which is copy memory interface node 1312, receives control information, and copy data, if copy data is forwarded by the source node. Since DMS copy engine 1311 has not sent any copy data, only control information is received.

Upon receipt of control information and possible receipt of copy data, the destination node may simply forward control information/copy data to the next copy ring node in the copy
ring 1300. Forwarding the control information/copy data in this way occurs when, in effect, the control information indicates that there is no read or write for a destination node to perform. Specifically, if the source address does not identify a memory address located in the DMS memory unit of the destination node and the write map does not indicate to write copy data to the DMD memory unit, or no copy data was forwarded from the source node along with the control information, there is no read or write to DMS memory for a destination node to perform. The destination node becomes the source node at 1510 and forwards control information/copy data to the next destination node.

[0359] Otherwise, one of two alternate sets of operations is performed in response to certain determinations as follows. First, at 1520, in response to a determination that the source address maps to the DMS memory unit ("source DMS memory unit") of the destination node, the destination node reads the copy data at the source address.

[0360] Second, at 1525, in response to a determination that the write map identifies the DMS memory unit of the destination node, the destination node writes, to the respective DMS memory unit, the copy data received, the copy data being written to the destination address specified by "Dest Addr" of the DMS-DMS descriptor. At 1530, the write map in the control data is set so that it no longer specifies to write copy data to that destination node.

[0361] After performing either operation 1520 or 1530, the destination node determines whether the write map specifies that the copy data is to be written to any DMS memory unit. If the determination is that write map specifies that copy data is to be written to any DMS memory unit, then the destination node becomes the source node at 1510 and transmits control information and/copy data to the next destination node. Otherwise, the internal DMS memory movement ends.

[0362] A DMS-DMS descriptor may specify a source address for copy data for a copy ring node that is not the first in copy ring node 1300, i.e. is not copy memory interface node 1312. In this case, only the control information is forwarded from copy ring node to copy ring node until the "copy data source node" is reached, that is, until the copy memory interface node that handles the DMS memory unit that corresponds to the source address is reached. The copy data source node reads the copy data from the respective DMS memory unit and forwards the copy data along with the control information to successive copy ring nodes.
It may be useful to shift data within a DMS memory unit. To accommodate this scenario, the DMS memory unit is identified, in a DMS-DMS descriptor, as both the source of copy data and a destination of copy data by the write map. The copy data is read from the source DMS memory unit by the source copy memory interface node, which then writes the copy data to the source DMS memory unit (which is also the destination DMS memory unit) at the destination address.

In another useful scenario, multiple core processors may each generate specific parts of a larger BV; each specific part is then distributed among multiple RID memory units to generate a copy of the larger BV in each of the multiple RID memory units. A copy of the BV may be efficiently assembled in each of the multiple of RID memory units using internal DMS memory movement. Each core processor may configure a DMS-DMS descriptor to load the respective BV part from the respective RID memory unit of the core processor into other RID memory units at a destination address corresponding to the respective BV part in the larger BV. The destination address to use is different for each core processor. Each core processor configures a DMS-DMS core processor to copy the BV part at the destination address in other RID memory units.

**BROADCASTING DATA TO MULTIPLE DMEM UNITS**

According to an embodiment, each data movement engine 130a, 130b, 130c, and 130d is connected via a corresponding DMAX to the DMEM units of a separate set of core processors served by the DMAX, and, in response to a single data descriptor that has a DMEM unit as a source or destination of a data movement, can only move data to and from the DMEMs connected to that DMAX. The set of core processors, the respective DMEM units, and the database movement engine are referred to as being local to each other. The other data movement engines, other core processors in Data Movement System 101 not in the set, and DMEM units of the other core processors are referred herein to as being remote.

For example, data movement engine 130a is connected by DMAX 110a to local DMEM units 102a and 102g of local core processors 103a and 103g. Data movement engine 130a may only move, in response to a single descriptor, data from main memory to either DMEM unit 102a or 102g. With respect to data movement engine 130a, DMEM units 105a and 105g of core processors 104a and 104g are referred to as being remote.
According to an embodiment, each of data movement engine 130a, 130b, 130c, and 130d, in response to a single descriptor with DMEM specified as the destination of the data movement, submitted by a local core processor, moves data from a source memory to multiple DMEM units, which may be both local and remote with respect to the data movement engine. For example, core processor 103a submits a data descriptor, which is processed by local data engine 130a. The data descriptor specifies to move data from main memory to multiple DMEM units, some of which are local to the data movement engine 130a and some of which are remote. In response, data movement engine 130a processes the descriptor, moving data from the source memory to the multiple DMEM units, as described in further detail below. Moving data to multiple DMEM units in response to a single descriptor that specifies so is referred to herein as a DMEM broadcast.

In a DMEM broadcast, data movement by a data movement engine to remote DMEM units is accomplished through a copy ring having copy memory interface nodes that are each connected to a data movement engine. Such a copy ring is depicted in FIG. 16.

Referring to FIG. 16, it depicts copy ring 1600. Like copy ring 1300, copy ring 1600 includes copy memory interface nodes. However, the copy memory interface nodes are each connected to a data movement engine and are each configured to write (or read) to circuitry of the data movement engine in response to control data and "broadcast data" transmitted over copy ring 1600. Each copy ring node in copy ring 1600 is linked by buses to two other copy ring nodes, thereby forming a loop or ring along which control data and broadcast data is forwarded among successive copy ring nodes, as described earlier for copy ring 1300.

Copy ring 1600 comprises copy memory interface node 1612, 1613, 1614, and 1615, which are connected to data movement engine 130a, data movement engine 130b, data movement engine 130c, and data movement engine 130d, respectively. DME copy engine 1611 comprises digital electronic circuitry configured to initiate transmission of control data and broadcast data over copy ring 1600 in response to receiving control data and broadcast data from a data movement engine 160.

A DMEM unit that is local to the data movement engine to which a copy memory interface node is connected is referred to as local with respect to the data movement engine. Thus, DMEM units 102a and 102g, which are local to data movement engine 130a, are local to copy memory interface node 1612.
DMEM BROADCAST DESCRIPTORS AND HANDLING

[0372] Data descriptors that may specify to perform a DMEM broadcast are referred to herein as DMEM broadcast descriptors. According to an embodiment, several types of data descriptors may be DMEM broadcast descriptors.

[0373] A descriptor type that specifies to move data from DDR memory to DMEM may specify to broadcast the data to one or more DMEM units. The DMEM units to which to broadcast are identified by a DMEM map. Similar to a write map, the DMEM map comprises a sequence of bits, each of which are associated with a DMEM unit and may be set to broadcast data to the DMEM unit.

[0374] The data descriptor is submitted by a requesting core processor as described previously for a data descriptor. The local data movement engine of the core processor retrieves the data from DDR memory and transmits the data to the DMEM unit of the requesting core processor to be written thereto.

[0375] The control information that is generated from the data descriptor includes the DMEM map. If the data movement engine determines that the DMEM map identifies any DMEM unit local to the data movement engine, the data movement engine treats data read from DDR memory as broadcast data and transmits the broadcast data to any local DMEM unit identified by the DMEM map.

[0376] If the data movement engine determines that the DMEM map identifies any DMEM unit that is remote to the data movement engine, the data movement engine treats the data read from DDR memory as broadcast data and transmits control data along with the broadcast data on copy ring 1600 to the next copy ring node on the ring. For example, if DME 103a was the local DME that received the data read from DDR, the copy ring node it is connected to, copy memory interface node 1612, places the control data and the broadcast data on ring 1600, where it is transmitted to copy memory interface node 1613.

[0377] The control data and broadcast data is forwarded along copy ring 1600 between successive copy memory interface nodes 1613, 1614, and 1615 similar to as described for copy ring 1300.

[0378] As each copy memory interface node receives the control data and broadcast data, the copy memory interface node determines whether the DMEM map identifies a DMEM unit local to the copy memory interface node. If the DMEM map identifies a DMEM unit local to the copy
memory interface node, the copy memory interface node writes control data and broadcast data to the internal circuitry of the data movement engine. The data movement engine transmits the broadcast data to any local DMEM unit identified by the DMEM map to be written thereto.

[0379] Similar to the write map, when broadcast data is written to the internal circuitry of a data movement engine, the respective copy memory interface node sets the DMEM map to indicate that broadcast data has been forwarded to the DMEM unit. Thus, if after copy memory interface node sets the DMEM map the DMEM map indicates that there are no DMEM units to which to forward broadcast data, copy memory interface node ceases to transmit the broadcast data.

[0380] The data descriptor may also identify tabular data manipulation operations (e.g. a scatter and a gather) to be performed. If the data movement engine determines that control data specifies to perform a tabular data manipulation operation, then a data movement engine moving broadcast data may transform the broadcast data according to the database operation and write broadcast data as transformed to a DMEM unit.

[0381] Not only may a data descriptor specify that the source of broadcast data is DDR memory, a data descriptor may specify the source of broadcast data is a DMEM unit or DMS memory. The DMEM units to which to broadcast are identified by a DMEM map in the data descriptor.

[0382] The data descriptor is submitted by a requesting core processor as described previously for a data descriptor. The local data movement engine of the core processor retrieves the broadcast data from the DMEM unit of the requesting core processor and transmits the control data generated for the data descriptor and the broadcast data on copy ring 1600 to the next copy ring node on the ring as described previously.

[0383] The control data and broadcast data is transmitted to and processed by all copy memory interface nodes (1312, 1613, 1614, and 1615) similar to as described above.

PERIPHERAL ACCESS TO COPY RING

[0384] A copy ring similar to those described above may be used to access DMS memory through a peripheral device. The peripheral device is connected to a copy ring similar to copy ring 1300. For example, the peripheral device may be a debug controller connected to the DMS copy engine. The debug controller may transmit control data to the copy ring engine, which transmits the control data along the copy ring. The source address specifies the DMS memory
unit from which to read data. The corresponding copy memory interface node reads data from
the DMS memory unit, and transmits the data along the copy ring to the copy engine, which
returns the data to the controller. Similarly, the control data can specify to copy the data to other
DMS memory units.

[0385] To broadcast data to multiple DMEM units, a copy ring similar to copy ring 1600
may be used. The peripheral device sends control data along with broadcast data to the copy
engine of a copy ring. The broadcast data is propagated by the copy ring to the DMEM units
specified in DMEM map via the respective data movement engines. The peripheral device may
be connected to the copy engine in the DMS by a master of a SOC ("system on a chip")
interconnect. The peripheral itself could be a network interface such as PCIe (Peripheral
Component Interconnect Express) or Ethernet. The control data may specify to perform tabular
data manipulation operations, which are performed by one or more database movement engines.

[0386] EXTENSIONS AND ALTERNATIVES

[0387] In the foregoing specification, embodiments of the invention have been described
with reference to numerous specific details that may vary from implementation to
implementation. Thus, the sole and exclusive indicator of what is the invention, and is intended
by the applicants to be the invention, is the set of claims that issue from this application, in the
specific form in which such claims issue, including any subsequent correction. Any definitions
expressly set forth herein for terms contained in such claims shall govern the meaning of such
terms as used in the claims. Hence, no limitation, element, property, feature, advantage or
attribute that is not expressly recited in a claim should limit the scope of such claim in any way.
The specification and drawings are, accordingly, to be regarded in an illustrative rather than a
restrictive sense.
CLAIMS

What is claimed is:

1. A method for moving tabular data to on-chip memory:
   in response to a particular memory location being pushed into a first register within a first register space that is accessible by a first set of electronic circuits:
   said first set of electronic circuits accessing a descriptor stored at the particular memory location, wherein the descriptor indicates:
   a width of a column of tabular data, a number of rows of said tabular data, and one or more tabular data manipulation operations to perform on the column of data;
   a source memory location for said tabular data
   a destination memory location for a data manipulation result of data manipulation operation;
   the first set of electronic circuits determining, based on the descriptor, control information indicating said one or more data manipulation operations to perform on the tabular data;
   the first set of electronic circuits transmitting, using a hardware data channel, the control information to a second set of electronic circuits to perform the one or more operations;
   according to the control information, said second set of electronic circuits retrieving said tabular data from said source memory location;
   applying said one or more data manipulation operations to said tabular data to generate said data manipulation result;
   causing said data manipulation result to be stored at said destination memory location.

2. The method of Claim 1, wherein the descriptor indicates a type of the descriptor.

3. The method of Claim 1, wherein the descriptor indicates a wait condition that should be satisfied prior to the step of the first set of electronic circuits determining the control information.

4. The method of Claim 1, further comprising:
   the first set of electronic circuits adding the descriptor to a hardware managed linked list;
wherein adding the descriptor to a hardware managed linked list comprises writing the particular memory location to a particular descriptor field of a second descriptor.

5. The method of Claim 1, wherein the descriptor indicates automatically incrementing source memory location for said tabular data and a source counter.

6. The method of Claim 5, further comprising:
   in response to a second memory location being pushed into the first register:
       said first set of electronic circuits accessing a second descriptor stored at the second memory location, wherein the second descriptor indicates:
           the source memory location for said tabular data;
   said second set of electronic circuits determining a new source memory location based at least in part on a value of the source counter.

7. The method of Claim 6, further comprising:
   first set of electronic circuits incrementing a source counter value by one after the step of the first set of electronic circuits transmitting the control information to the second set of electronic circuits.

8. The method of Claim 1, further comprising:
   in response to a second memory location being pushed into the first register:
       said first set of electronic circuits accessing a second descriptor stored at the second memory location, wherein the second descriptor indicates:
           the particular memory location;
   said first set of electronic circuits accessing the descriptor stored at the particular memory location;
   said first set of electronic circuits determining, based on the descriptor, control information indicating said one or more data manipulation operations to perform on the tabular data.

9. The method of Claim 8, further comprising:
   the second descriptor indicates an iteration count;
   the first set of electronic circuits determining that the iteration count is not zero prior to accessing the descriptor stored at the particular memory location.

10. The method of Claim 1, wherein the descriptor indicates a notification flag.

11. A method comprising:
a data processing core coupled with a first on-chip memory pushing a particular memory
address of a descriptor into a first register within a first register space that is
accessible by a first set of electronic circuits that is coupled to said on-chip
memory;
wherein the descriptor indicates a width of a column of tabular data, a number of
rows of said tabular data, and a data manipulation operation to perform on
said tabular data;
in response to said particular memory address of said descriptor being pushed into said
first register by said data processing core:
the first set of electronic circuits, using the particular memory address in the first
register, accessing the descriptor in the first on-chip memory;
the first set of electronic circuits determining, based on the descriptor, control
information indicating one or more data manipulation operations to perform;
in response to determining the control information:
the first set of electronic circuits transmitting, using a hardware data channel, the
control information to a second set of electronic circuits to perform the
one or more data manipulation operations.

12. The method of Claim 11, further comprising:
the data processing core determining that said particular memory address is available in a
second register within a second register space that is designated to store memory
addresses of the first memory that are free; and
in response to the particular memory address being available in the second register,
the data processing core storing the descriptor at the particular memory address.

13. The method of Claim 11, further comprising:
wherein the data processing core determines whether a flag, indicating that one or more
memory addresses of the first memory is available in registers within a second
register space, is set, wherein the second register space is designated to store
memory addresses of the first memory that are free;
in response to the flag being set:
the data processing core determining that the particular memory address is available in a particular register within the register space.

14. The method of Claim 11, further comprising:
   the first set of electronic circuits adding the descriptor to a linked list managed by hardware of a data movement system.

15. The method of Claim 14, wherein adding the descriptor to the linked list is performed by specifying in a link address field of the descriptor a memory address of a second descriptor that is different from the first descriptor, wherein the second descriptor is in a second register within the first register space.
FIG. 2
FIG. 4
In response to a particular memory location being pushed into a first register within a first register space that is accessible by a first set of electronic circuits,

Said first set of electronic circuits accesses a descriptor stored at the particular memory location

The first set of electronic circuits determine, based on the descriptor, one or more instructions indicating said one or more data manipulation operations to perform on the tabular data

The first set of electronic circuits transmit, using a hardware data channel, the one or more instructions to a second set of electronic circuits to perform the one or more operations

According to the one or more instructions, said second set of electronic circuits retrieve said tabular data from a source memory location

Apply said one or more data manipulation operations to said tabular data to generate a data manipulation result

Cause said data manipulation result to be stored at said destination location.
DMS LOAD STAGE (move columns of rows being partitioned to DMS memory) 721

CID GENERATION STAGE (generate core identifiers that for each row, the core identifiers specifying where to partition row) 722

CORE PARTITIONING STAGE (partition rows according to the respective core identifiers) 723
"Notify core processor when sub-buffer at current buffer index is full"

"Move current buffer index when core issues notification that sub-buffer at the current buffer ptr has been processed"

COLUMN FIFO BUFFER 802a

FIG. 8A
FIG. 9B
FIG. 11A
FIG. 14
INTERNATIONAL SEARCH REPORT

A. CLASSIFICATION OF SUBJECT MATTER

INV. G06F13/28

ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

EPO-Internal, WPI Data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>US 5 175 810 A (YOUNG CAROL A [US] ET AL) 29 December 1992 (1992-12-29) column 1, line 60 - column 2, line 60 figure 1</td>
<td>1-15</td>
</tr>
</tbody>
</table>

* Special categories of cited documents:

“A” document defining the general state of the art which is not considered to be of particular relevance

“E” earlier application or patent but published on or after the international filing date

“L” document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

“O” document referring to an oral disclosure, use, exhibition or other means

“P” document published prior to the international filing date but later than the priority date claimed

“T” later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

“X” document of particular relevance: the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

“Y” document of particular relevance: the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

“A” document member of the same patent family

Date of the actual completion of the international search: 13 June 2017

Date of mailing of the international search report: 27/06/2017

Name and mailing address of the ISA:

European Patent Office, P.B. 5818 Patentlaan 2 NL-2280 HV Rijswijk

Tel. (+31-70) 340-2040, Fax. (+31-70) 340-3016

Authorized officer: Nguyen Xuan Hi ep, C

See patent family annex.

Form PCT/ISA/210 (second sheet) (April 2005)
<table>
<thead>
<tr>
<th>Patent document cited in search report</th>
<th>Publication date</th>
<th>Patent family member(s)</th>
<th>Publication date</th>
</tr>
</thead>
<tbody>
<tr>
<td>US 2006179255 A1</td>
<td>10-08-2006</td>
<td>Wo 2006085665 A2</td>
<td>17-08-2006</td>
</tr>
<tr>
<td>US 5280575 A</td>
<td>18-01-1994</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>