US008639882B2

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

(10) **Patent No.: US 8,639,882 B2**
(45) **Date of Patent: Jan. 28, 2014**

(54) **METHODS AND APPARATUS FOR SOURCE OPERAND COLLECTOR CACHING**

(75) Inventors: **Jack Hilaire Choquette**, Palo Alto, CA (US); **Manuel Olivier Gautho**, Los Gatos, CA (US); **John Erik Lindholm**, Saratoga, CA (US)

(73) Assignee: **Nvidia Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 215 days.

(21) Appl. No.: **13/326,183**

(22) Filed: **Dec. 14, 2011**

(65) **Prior Publication Data**

US 2013/0159628 A1 Jun. 20, 2013

(51) **Int. Cl.**
G06F 12/00 (2006.01)

(52) **U.S. Cl.**
USPC **711/126**

(58) **Field of Classification Search**
None
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

8,200,949 B1 * 6/2012 Tarjan et al. 712/225
2011/0072243 A1 * 3/2011 Qiu et al. 712/214

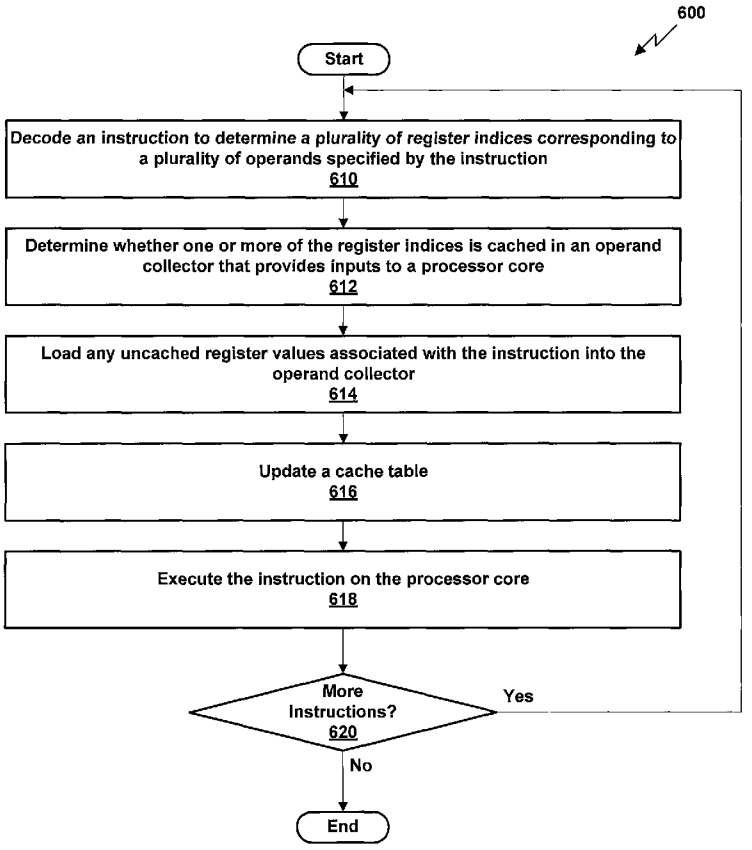
* cited by examiner

Primary Examiner — Hiep Nguyen
(74) *Attorney, Agent, or Firm* — Patterson + Sheridan, L.L.P.

(57) **ABSTRACT**

Methods and apparatus for source operand collector caching. In one embodiment, a processor includes a register file that may be coupled to storage elements (i.e., an operand collector) that provide inputs to the datapath of the processor core for executing an instruction. In order to reduce bandwidth between the register file and the operand collector, operands may be cached and reused in subsequent instructions. A scheduling unit maintains a cache table for monitoring which register values are currently stored in the operand collector. The scheduling unit may also configure the operand collector to select the particular storage elements that are coupled to the inputs to the datapath for a given instruction.

19 Claims, 8 Drawing Sheets



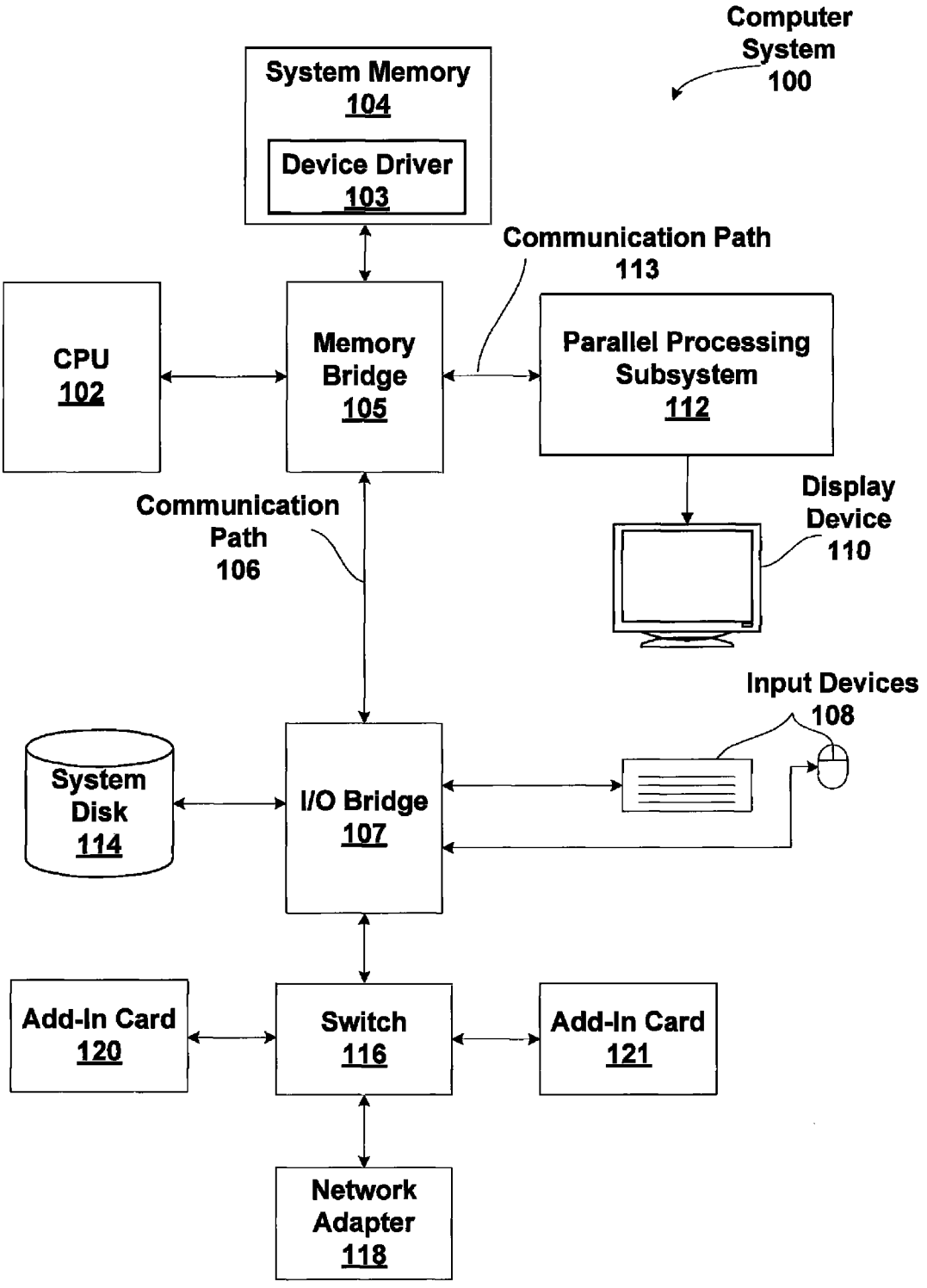

Computer System 100
System Memory 104
Device Driver 103
Communication Path 113
CPU 102
Memory Bridge 105
Parallel Processing Subsystem 112
Display Device 110
Communication Path 106
Input Devices 108
System Disk 114
I/O Bridge 107
Add-In Card 120
Switch 116
Add-In Card 121
Network Adapter 118

Figure 1

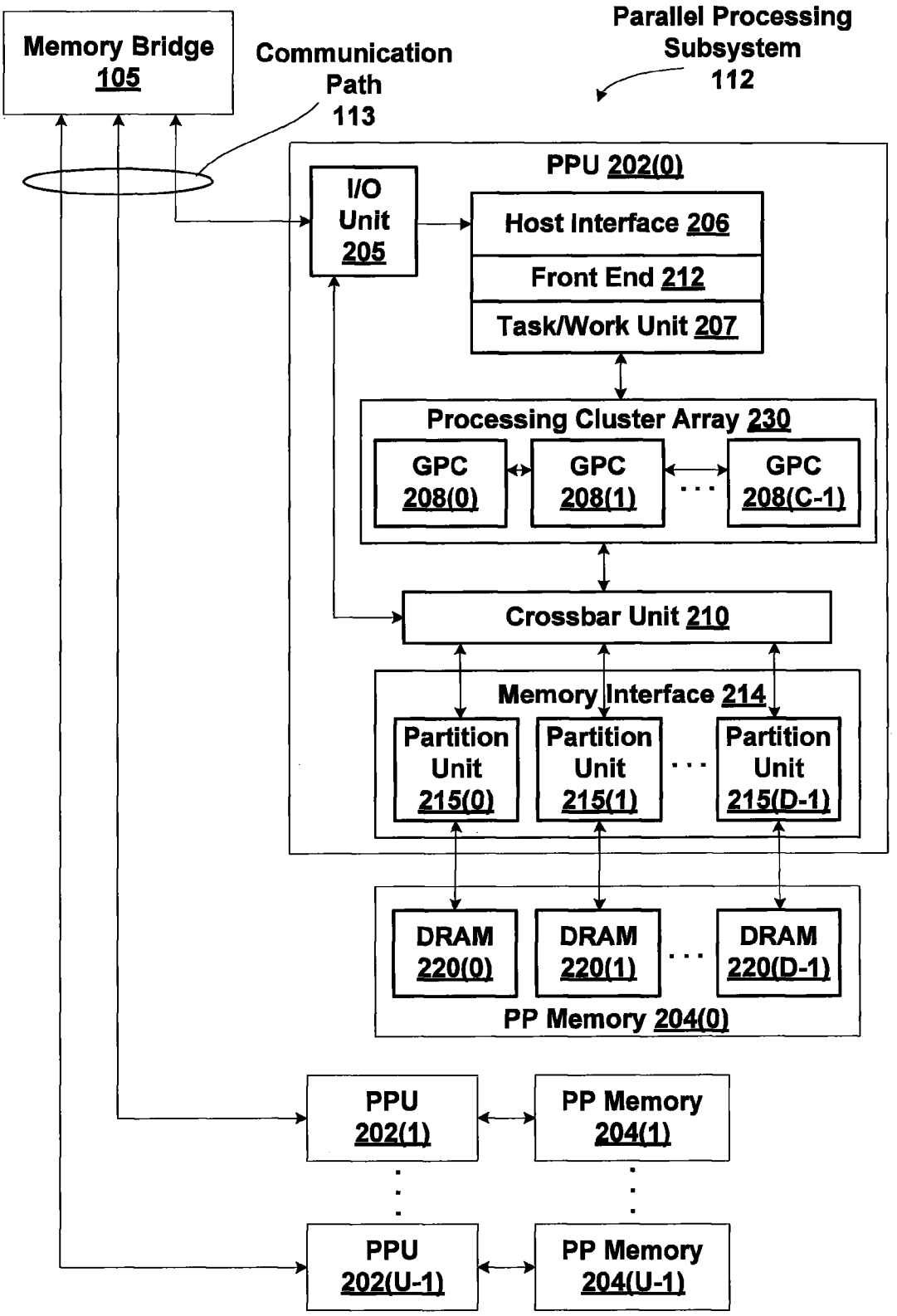

Memory Bridge 105
Communication Path 113
Parallel Processing Subsystem 112
PPU 202(0)
I/O Unit 205
Host Interface 206
Front End 212
Task/Work Unit 207
Processing Cluster Array 230
GPC 208(0)
GPC 208(1)
GPC 208(C-1)
Crossbar Unit 210
Memory Interface 214
Partition Unit 215(0)
Partition Unit 215(1)
Partition Unit 215(D-1)
DRAM 220(0)
DRAM 220(1)
DRAM 220(D-1)
PP Memory 204(0)
PPU 202(1)
PP Memory 204(1)
PPU 202(U-1)
PP Memory 204(U-1)

Figure 2

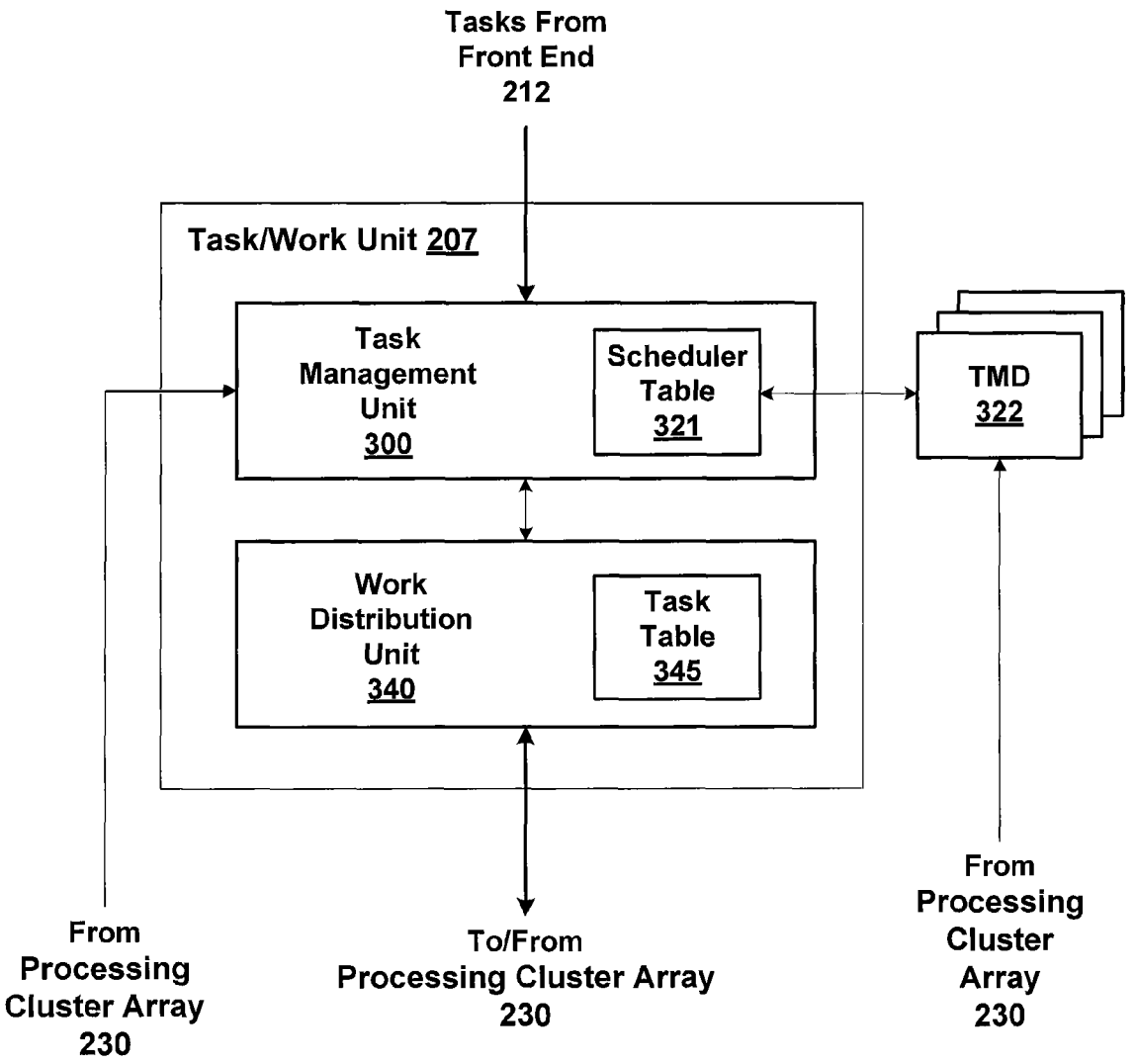

Tasks From
Front End
212
Task/Work Unit 207
Task
Management
Unit
300
Scheduler
Table
321
TMD
322
Work
Distribution
Unit
340
Task
Table
345
From
Processing
Cluster Array
230
To/From
Processing Cluster Array
230
From
Processing
Cluster
Array
230

Figure 3A

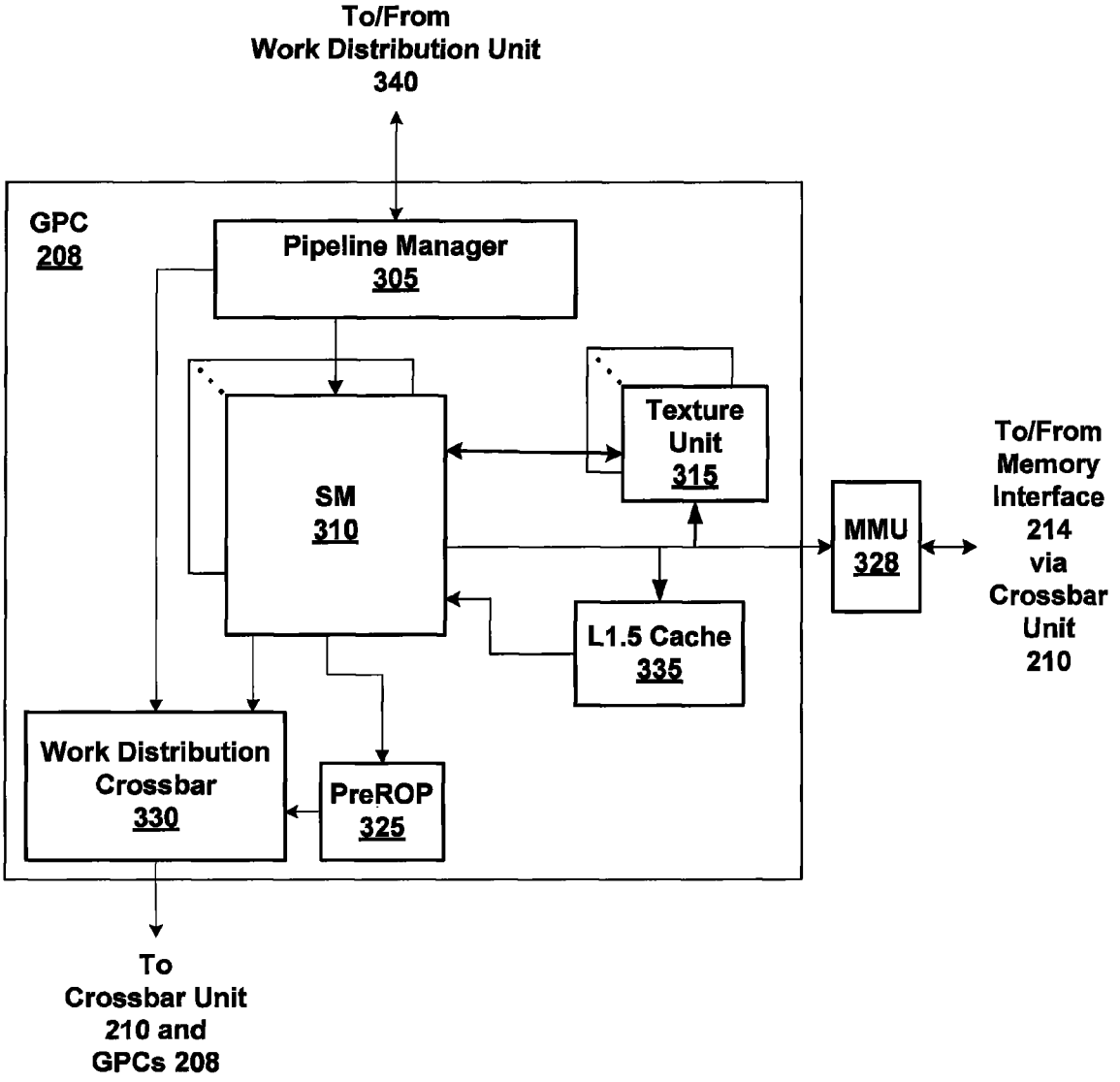

To/From
Work Distribution Unit
340
GPC
208
Pipeline Manager
305
SM
310
Texture
Unit
315
MMU
328
To/From
Memory
Interface
214
via
Crossbar
Unit
210
L1.5 Cache
335
Work Distribution
Crossbar
330
PreROP
325
To
Crossbar Unit
210 and
GPCs 208

Figure 3B

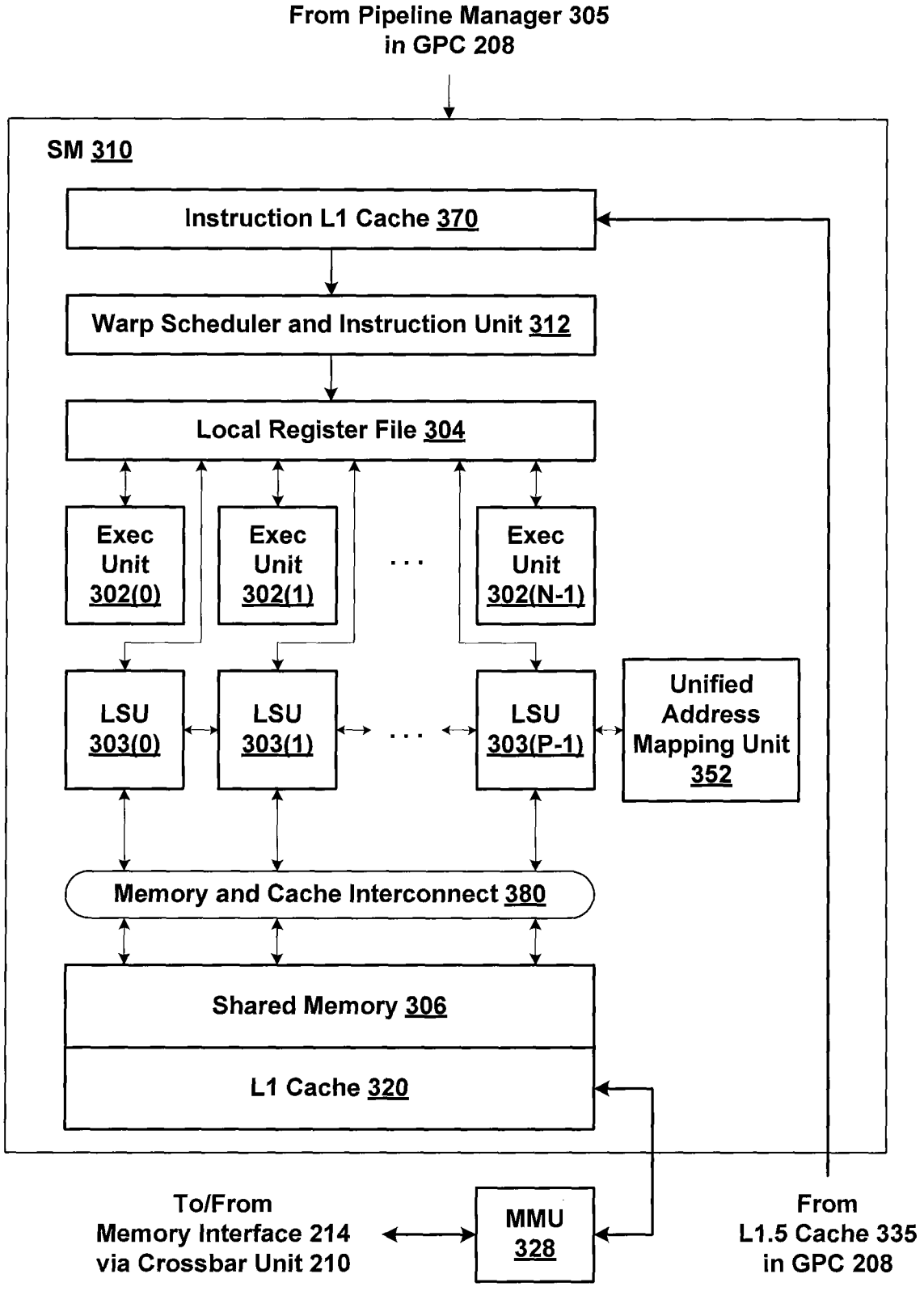

From Pipeline Manager 305
in GPC 208
SM 310
Instruction L1 Cache 370
Warp Scheduler and Instruction Unit 312
Local Register File 304
Exec Unit 302(0)
Exec Unit 302(1)
Exec Unit 302(N-1)
LSU 303(0)
LSU 303(1)
LSU 303(P-1)
Unified Address Mapping Unit 352
Memory and Cache Interconnect 380
Shared Memory 306
L1 Cache 320
To/From Memory Interface 214 via Crossbar Unit 210
MMU 328
From L1.5 Cache 335 in GPC 208

Figure 3C

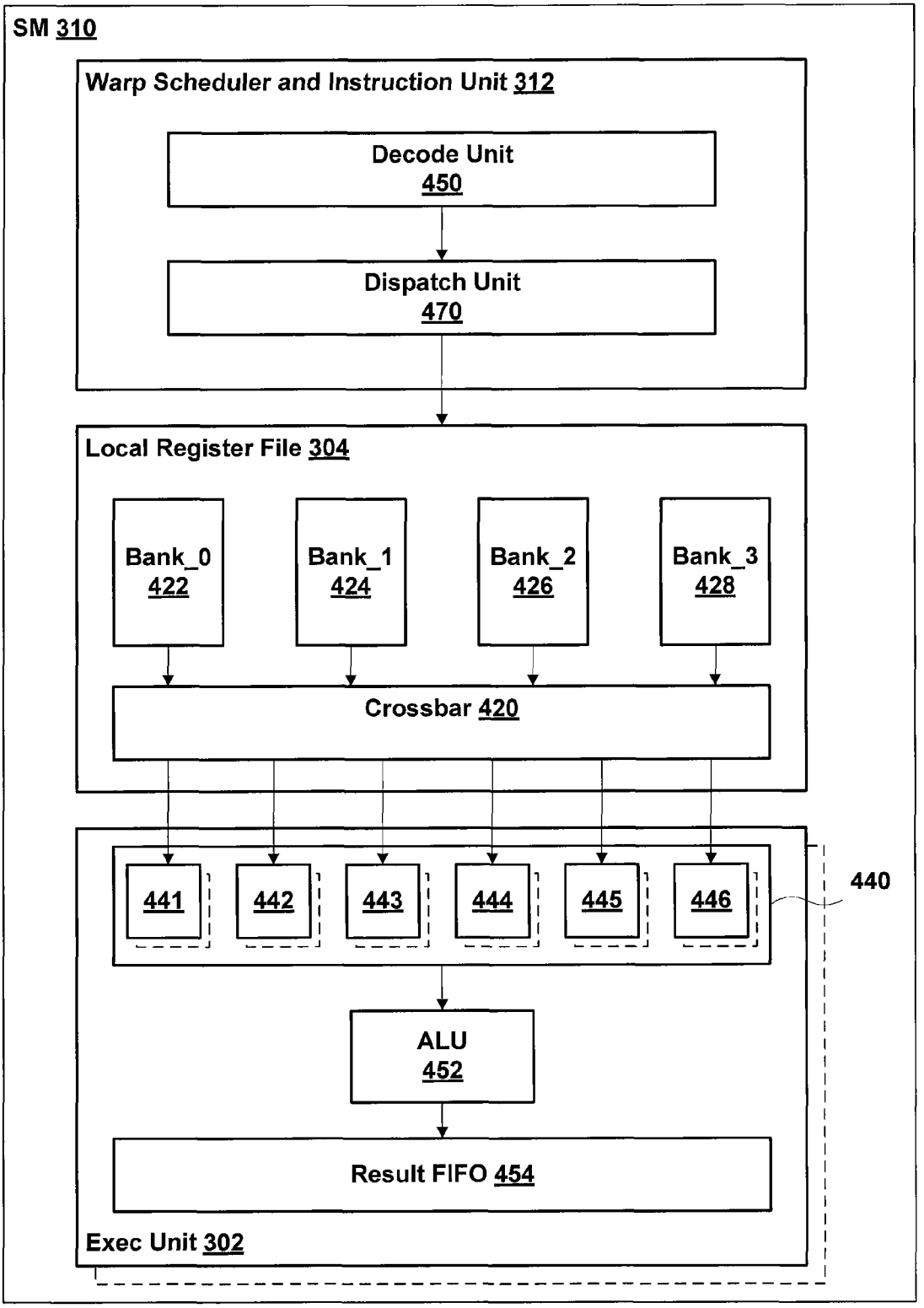

SM 310
Warp Scheduler and Instruction Unit 312
Decode Unit 450
Dispatch Unit 470
Local Register File 304
Bank_0 422
Bank_1 424
Bank_2 426
Bank_3 428
Crossbar 420
441
442
443
444
445
446
440
ALU 452
Result FIFO 454
Exec Unit 302

Figure 4

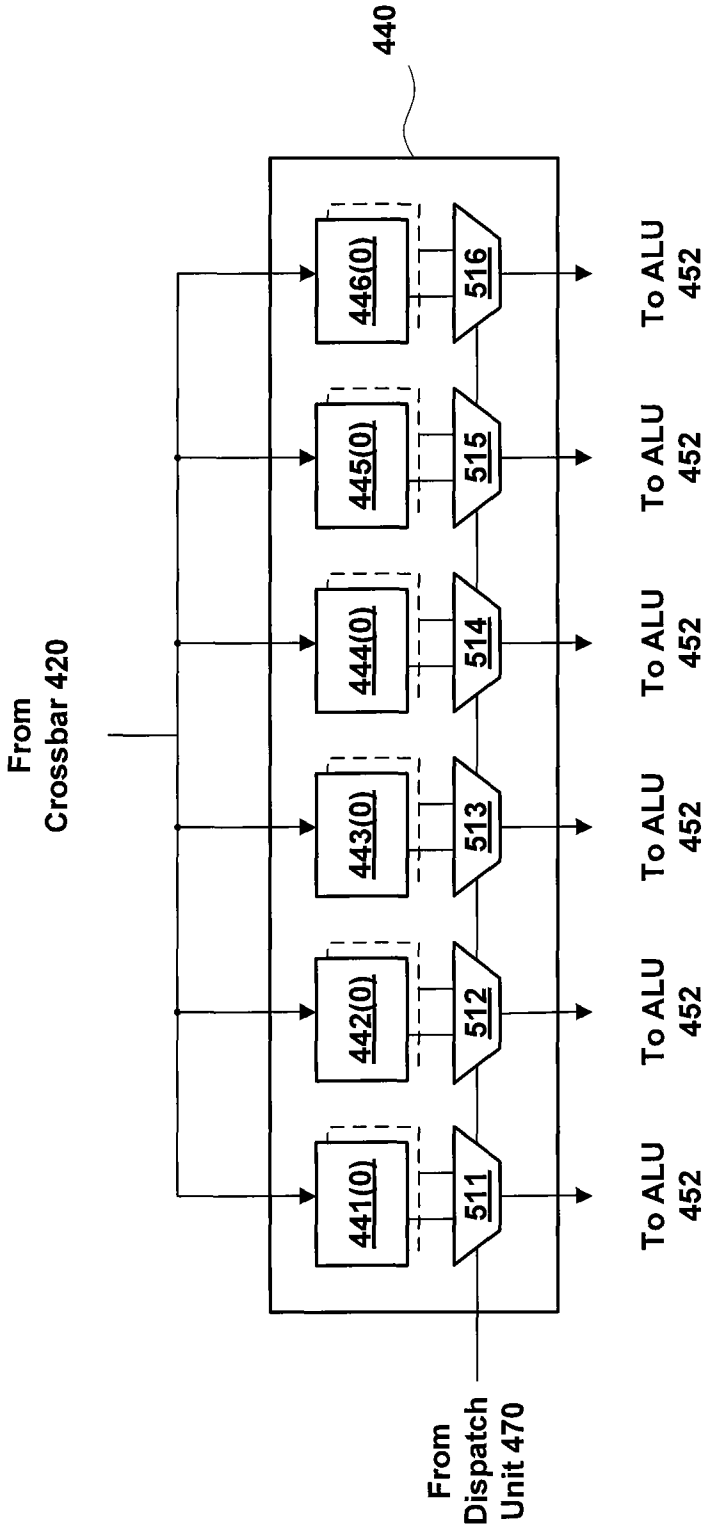

440
From Crossbar 420
From Dispatch Unit 470
441(0)
442(0)
443(0)
444(0)
445(0)
446(0)
511
512
513
514
515
516
To ALU 452
To ALU 452
To ALU 452
To ALU 452
To ALU 452
To ALU 452

Figure 5

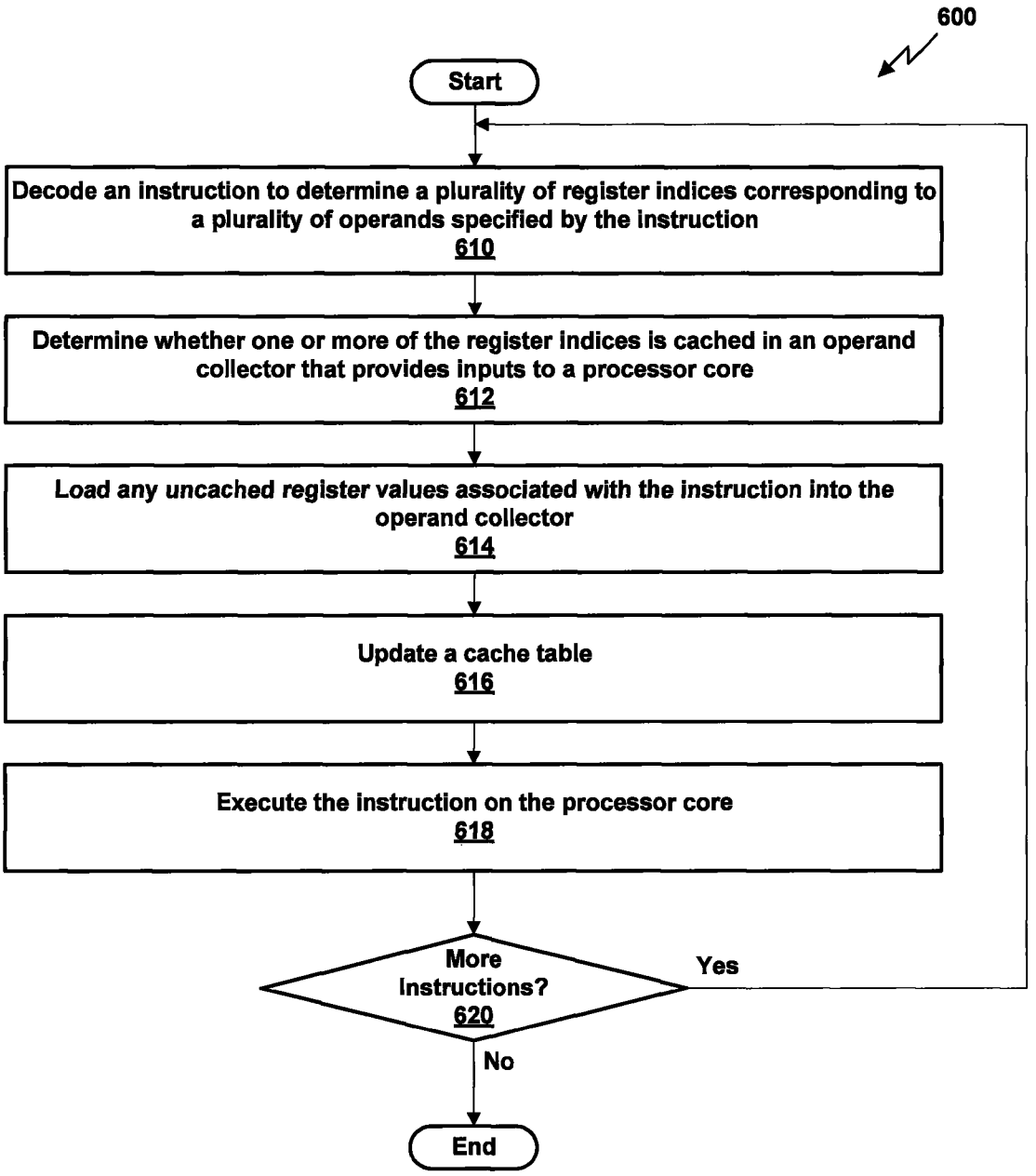

600
Start
Decode an instruction to determine a plurality of register indices corresponding to a plurality of operands specified by the instruction
610
Determine whether one or more of the register indices is cached in an operand collector that provides inputs to a processor core
612
Load any uncached register values associated with the instruction into the operand collector
614
Update a cache table
616
Execute the instruction on the processor core
618
More Instructions?
620
Yes
No
End

Figure 6

METHODS AND APPARATUS FOR SOURCE OPERAND COLLECTOR CACHING

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present disclosure generally relates to processor registers, and more specifically to methods and apparatus for source operand collector caching.

2. Description of the Related Art

Parallel processors have multiple independent cores that enable multiple threads to be executed simultaneously using different hardware resources. SIMD (single instruction, multiple data) architecture processors execute the same instruction on each of the multiple cores where each core executes on different input data. MIMD (multiple instruction, multiple data) architecture processors execute different instructions on different cores with different input data supplied to each core. Parallel processors may also be multi-threaded, which enables two or more threads to execute substantially simultaneously using the resources of a single processing core (i.e., the different threads are executed on the core during different clock cycles).

When a processor schedules an instruction for execution by a processor core, the processor writes certain values into special registers in a register file coupled to the processor core. One register may store the opcode that specifies the operation to be performed by the processor core and additional registers may store operand values used as input to the processor core for executing the instruction. In order for an operation to be executed, each of the values must be written into the register file and then coupled to the inputs of the datapath via a crossbar or other data transmission means. Each instruction may require new registers in the register file to be connected to the inputs at the top of the datapath.

One problem with the above architectures is that configuring the crossbar to couple register values stored in the register file to the inputs at the top of the datapath requires one or more clock cycles to perform. The time required to load each operand introduces latencies into the overall processing efficiency. Furthermore, the crossbar may be configured so that only one operand may be coupled to the inputs of the datapath during each clock cycle.

Accordingly, what is needed in the art is an improved technique for loading values from the register file into the inputs of a datapath of a processor core.

SUMMARY OF THE INVENTION

One embodiment sets forth a method for executing instructions on a processor core implemented with source operand collector caching. The method includes the steps of decoding an instruction to determine a plurality of operands specified by the instruction, and, for each operand in the plurality of operands, determining that a particular operand is not stored in a cache coupled to inputs of a datapath within the processor core, loading the particular operand into the cache from a local register file associated with the processor core. The method further includes the step of configuring the processor core to execute the instruction using operands stored in the cache.

Another embodiment sets forth a computer-readable storage medium including instructions. When the instructions are executed by a processor core, the instructions cause the processor core to perform the steps of decoding an instruction to determine a plurality of operands specified by the instruction, and, for each operand in the plurality of operands, determining that a particular operand is not stored in a cache coupled to inputs of a datapath within the processor core, loading the particular operand into the cache from a local register file associated with the processor core. The steps also include configuring the processor core to execute the instruction using operands stored in the cache.

Yet another embodiment sets forth a system that includes a processor core having a datapath for executing instructions, a cache coupled to the inputs of the datapath, and a scheduling unit. The scheduling unit is configured to decode an instruction to determine a plurality of operands specified by the instruction, and, for each operand in the plurality of operands, determining that a particular operand is not stored in a cache coupled to inputs of a datapath within the processor core, load the particular operand into the cache from a local register file associated with the processor core. The scheduling unit is also configured to execute the instruction using the operands stored in the cache.

BRIEF DESCRIPTION OF THE DRAWINGS

So that the manner in which the above recited features of the present disclosure can be understood in detail, a more particular description, briefly summarized above, may be had by reference to example embodiments, some of which are illustrated in the appended drawings. It is to be noted, however, that the appended drawings illustrate only typical embodiments of this disclosure and are therefore not to be considered limiting of its scope, for the disclosure may admit to other equally effective embodiments.

FIG. **1** is a block diagram illustrating a computer system configured to implement one or more aspects of the present disclosure;

FIG. **2** is a block diagram of a parallel processing subsystem for the computer system of FIG. **1**, according to one embodiment of the present disclosure;

FIG. **3**A is a block diagram of the front end of FIG. **2**, according to one embodiment of the present disclosure;

FIG. **3**B is a block diagram of a general processing cluster within one of the parallel processing units of FIG. **2**, according to one embodiment of the present disclosure;

FIG. **3**C is a block diagram of a portion of the streaming multiprocessor of FIG. **3**B, according to one embodiment of the present disclosure;

FIG. **4** is a block diagram of a portion of the streaming multiprocessor of FIG. **3**B, according to another example embodiment of the present disclosure;

FIG. **5** illustrates the operand collector of FIG. **4**, according to one example embodiment of the present disclosure; and

FIG. **6** is a flow chart illustrating a method for executing instructions on a processor core implemented with source operand collector caching, according to one example embodiment of the present disclosure.

DETAILED DESCRIPTION

In the following description, numerous specific details are set forth to provide a more thorough understanding of the present disclosure. However, it will be apparent to one of skill in the art that the present disclosure may be practiced without one or more of these specific details.

The disclosure describes methods and apparatus for source operand collector caching. In one embodiment, a processor includes a register file that may be coupled to storage elements (i.e., an operand collector) that provide inputs to the datapath of the processor core for executing an instruction. In order to reduce bandwidth between the register file and the

operand collector, operands may be cached and reused in subsequent instructions. Consequently, only a subset of the operands specified by a given instruction may need to be loaded into the operand collector. A scheduling unit maintains a cache table for monitoring the register values currently stored in the operand collector. The scheduling unit may also configure the operand collector to select the particular storage elements that are coupled to the inputs to the datapath for a given instruction, allowing operands for two or more instructions to be cached concurrently.

System Overview

FIG. **1** is a block diagram illustrating a computer system **100** configured to implement one or more aspects of the present disclosure. Computer system **100** includes a central processing unit (CPU) **102** and a system memory **104** communicating via an interconnection path that may include a memory bridge **105**. Memory bridge **105**, which may be, e.g., a Northbridge chip, is connected via a bus or other communication path **106** (e.g., a HyperTransport link) to an I/O (input/output) bridge **107**. I/O bridge **107**, which may be, e.g., a Southbridge chip, receives user input from one or more user input devices **108** (e.g., keyboard, mouse) and forwards the input to CPU **102** via communication path **106** and memory bridge **105**. A parallel processing subsystem **112** is coupled to memory bridge **105** via a bus or second communication path **113** (e.g., a Peripheral Component Interconnect (PCI) Express, Accelerated Graphics Port, or HyperTransport link); in one embodiment parallel processing subsystem **112** is a graphics subsystem that delivers pixels to a display device **110** (e.g., a conventional cathode ray tube or liquid crystal display based monitor). A system disk **114** is also connected to I/O bridge **107**. A switch **116** provides connections between I/O bridge **107** and other components such as a network adapter **118** and various add-in cards **120** and **121**. Other components (not explicitly shown), including universal serial bus (USB) or other port connections, compact disc (CD) drives, digital video disc (DVD) drives, film recording devices, and the like, may also be connected to I/O bridge **107**. The various communication paths shown in FIG. **1**, including the specifically named communications paths **106** and **113**, may be implemented using any suitable protocols, such as PCI Express, AGP (Accelerated Graphics Port), HyperTransport, or any other bus or point-to-point communication protocol(s), and connections between different devices may use different protocols as is known in the art.

In one embodiment, the parallel processing subsystem **112** incorporates circuitry optimized for graphics and video processing, including, for example, video output circuitry, and constitutes a graphics processing unit (GPU). In another embodiment, the parallel processing subsystem **112** incorporates circuitry optimized for general purpose processing, while preserving the underlying computational architecture, described in greater detail herein. In yet another embodiment, the parallel processing subsystem **112** may be integrated with one or more other system elements in a single subsystem, such as joining the memory bridge **105**, CPU **102**, and I/O bridge **107** to form a system on chip (SoC).

It will be appreciated that the system shown herein is illustrative and that variations and modifications are possible. The connection topology, including the number and arrangement of bridges, the number of CPUs **102**, and the number of parallel processing subsystems **112**, may be modified as desired. For instance, in some embodiments, system memory **104** is connected to CPU **102** directly rather than through a bridge, and other devices communicate with system memory **104** via memory bridge **105** and CPU **102**. In other alternative topologies, parallel processing subsystem **112** is connected to I/O bridge **107** or directly to CPU **102**, rather than to memory bridge **105**. In still other embodiments, I/O bridge **107** and memory bridge **105** might be integrated into a single chip instead of existing as one or more discrete devices. Large embodiments may include two or more CPUs **102** and two or more parallel processing systems **112**. The particular components shown herein are optional; for instance, any number of add-in cards or peripheral devices might be supported. In some embodiments, switch **116** is eliminated, and network adapter **118** and add-in cards **120**, **121** connect directly to I/O bridge **107**.

FIG. **2** illustrates a parallel processing subsystem **112**, according to one embodiment of the present disclosure. As shown, parallel processing subsystem **112** includes one or more parallel processing units (PPUs) **202**, each of which is coupled to a local parallel processing (PP) memory **204**. In general, a parallel processing subsystem includes a number U of PPUs, where U≥1. (Herein, multiple instances of like objects are denoted with reference numbers identifying the object and parenthetical numbers identifying the instance where needed.) PPUs **202** and parallel processing memories **204** may be implemented using one or more integrated circuit devices, such as programmable processors, application specific integrated circuits (ASICs), or memory devices, or in any other technically feasible fashion.

Referring again to FIG. **1** as well as FIG. **2**, in some embodiments, some or all of PPUs **202** in parallel processing subsystem **112** are graphics processors with rendering pipelines that can be configured to perform various operations related to generating pixel data from graphics data supplied by CPU **102** and/or system memory **104** via memory bridge **105** and the second communication path **113**, interacting with local parallel processing memory **204** (which can be used as graphics memory including, e.g., a conventional frame buffer) to store and update pixel data, delivering pixel data to display device **110**, and the like. In some embodiments, parallel processing subsystem **112** may include one or more PPUs **202** that operate as graphics processors and one or more other PPUs **202** that are used for general-purpose computations. The PPUs may be identical or different, and each PPU may have a dedicated parallel processing memory device(s) or no dedicated parallel processing memory device(s). One or more PPUs **202** in parallel processing subsystem **112** may output data to display device **110** or each PPU **202** in parallel processing subsystem **112** may output data to one or more display devices **110**.

In operation, CPU **102** is the master processor of computer system **100**, controlling and coordinating operations of other system components. In particular, CPU **102** issues commands that control the operation of PPUs **202**. In some embodiments, CPU **102** writes a stream of commands for each PPU **202** to a data structure (not explicitly shown in either FIG. **1** or FIG. **2**) that may be located in system memory **104**, parallel processing memory **204**, or another storage location accessible to both CPU **102** and PPU **202**. A pointer to each data structure is written to a pushbuffer to initiate processing of the stream of commands in the data structure. The PPU **202** reads command streams from one or more pushbuffers and then executes commands asynchronously relative to the operation of CPU **102**. Execution priorities may be specified for each pushbuffer by an application program via the device driver **103** to control scheduling of the different pushbuffers.

Referring back now to FIG. **2** as well as FIG. **1**, each PPU **202** includes an I/O (input/output) unit **205** that communicates with the rest of computer system **100** via communica-

tion path **113**, which connects to memory bridge **105** (or, in one alternative embodiment, directly to CPU **102**). The connection of PPU **202** to the rest of computer system **100** may also be varied. In some embodiments, parallel processing subsystem **112** is implemented as an add-in card that can be inserted into an expansion slot of computer system **100**. In other embodiments, a PPU **202** can be integrated on a single chip with a bus bridge, such as memory bridge **105** or I/O bridge **107**. In still other embodiments, some or all elements of PPU **202** may be integrated on a single chip with CPU **102**.

In one embodiment, communication path **113** is a PCI Express link, in which dedicated lanes are allocated to each PPU **202**, as is known in the art. Other communication paths may also be used. An I/O unit **205** generates packets (or other signals) for transmission on communication path **113** and also receives all incoming packets (or other signals) from communication path **113**, directing the incoming packets to appropriate components of PPU **202**. For example, commands related to processing tasks may be directed to a host interface **206**, while commands related to memory operations (e.g., reading from or writing to parallel processing memory **204**) may be directed to a memory crossbar unit **210**. Host interface **206** reads each pushbuffer and outputs the command stream stored in the pushbuffer to a front end **212**.

Each PPU **202** advantageously implements a highly parallel processing architecture. As shown in detail, PPU **202(0)** includes a processing cluster array **230** that includes a number C of general processing clusters (GPCs) **208**, where C≥1. Each GPC **208** is capable of executing a large number (e.g., hundreds or thousands) of threads concurrently, where each thread is an instance of a program. In various applications, different GPCs **208** may be allocated for processing different types of programs or for performing different types of computations. The allocation of GPCs **208** may vary dependent on the workload arising for each type of program or computation.

GPCs **208** receive processing tasks to be executed from a work distribution unit within a task/work unit **207**. The work distribution unit receives pointers to processing tasks that are encoded as task metadata (TMD) and stored in memory. The pointers to TMDs are included in the command stream that is stored as a pushbuffer and received by the front end unit **212** from the host interface **206**. Processing tasks that may be encoded as TMDs include indices of data to be processed, as well as state parameters and commands defining how the data is to be processed (e.g., what program is to be executed). The task/work unit **207** receives tasks from the front end **212** and ensures that GPCs **208** are configured to a valid state before the processing specified by each one of the TMDs is initiated. A priority may be specified for each TMD that is used to schedule execution of the processing task. Processing tasks can also be received from the processing cluster array **230**. Optionally, the TMD can include a parameter that controls whether the TMD is added to the head or the tail for a list of processing tasks (or list of pointers to the processing tasks), thereby providing another level of control over priority.

Memory interface **214** includes a number D of partition units **215** that are each directly coupled to a portion of parallel processing memory **204**, where D≥1. As shown, the number of partition units **215** generally equals the number of dynamic random access memory (DRAM) **220**. In other embodiments, the number of partition units **215** may not equal the number of memory devices. Persons of ordinary skill in the art will appreciate that DRAM **220** may be replaced with other suitable storage devices and can be of generally conventional design. A detailed description is therefore omitted. Render targets, such as frame buffers or texture maps may be stored across DRAMs **220**, allowing partition units **215** to write portions of each render target in parallel to efficiently use the available bandwidth of parallel processing memory **204**.

Any one of GPCs **208** may process data to be written to any of the DRAMs **220** within parallel processing memory **204**. Crossbar unit **210** is configured to route the output of each GPC **208** to the input of any partition unit **215** or to another GPC **208** for further processing. GPCs **208** communicate with memory interface **214** through crossbar unit **210** to read from or write to various external memory devices. In one embodiment, crossbar unit **210** has a connection to memory interface **214** to communicate with I/O unit **205**, as well as a connection to local parallel processing memory **204**, thereby enabling the processing cores within the different GPCs **208** to communicate with system memory **104** or other memory that is not local to PPU **202**. In the embodiment shown in FIG. **2**, crossbar unit **210** is directly connected with I/O unit **205**. Crossbar unit **210** may use virtual channels to separate traffic streams between the GPCs **208** and partition units **215**.

Again, GPCs **208** can be programmed to execute processing tasks relating to a wide variety of applications, including but not limited to, linear and nonlinear data transforms, filtering of video and/or audio data, modeling operations (e.g., applying laws of physics to determine position, velocity and other attributes of objects), image rendering operations (e.g., tessellation shader, vertex shader, geometry shader, and/or pixel shader programs), and so on. PPUs **202** may transfer data from system memory **104** and/or local parallel processing memories **204** into internal (on-chip) memory, process the data, and write result data back to system memory **104** and/or local parallel processing memories **204**, where such data can be accessed by other system components, including CPU **102** or another parallel processing subsystem **112**.

A PPU **202** may be provided with any amount of local parallel processing memory **204**, including no local memory, and may use local memory and system memory in any combination. For instance, a PPU **202** can be a graphics processor in a unified memory architecture (UMA) embodiment. In such embodiments, little or no dedicated graphics (parallel processing) memory would be provided, and PPU **202** would use system memory exclusively or almost exclusively. In UMA embodiments, a PPU **202** may be integrated into a bridge chip or processor chip or provided as a discrete chip with a high-speed link (e.g., PCI Express) connecting the PPU **202** to system memory via a bridge chip or other communication means.

As noted above, any number of PPUs **202** can be included in a parallel processing subsystem **112**. For instance, multiple PPUs **202** can be provided on a single add-in card, or multiple add-in cards can be connected to communication path **113**, or one or more of PPUs **202** can be integrated into a bridge chip. PPUs **202** in a multi-PPU system may be identical to or different from one another. For instance, different PPUs **202** might have different numbers of processing cores, different amounts of local parallel processing memory, and so on. Where multiple PPUs **202** are present, those PPUs may be operated in parallel to process data at a higher throughput than is possible with a single PPU **202**. Systems incorporating one or more PPUs **202** may be implemented in a variety of configurations and form factors, including desktop, laptop, or handheld personal computers, servers, workstations, game consoles, embedded systems, and the like.

Multiple Concurrent Task Scheduling

Multiple processing tasks may be executed concurrently on the GPCs **208** and a processing task may generate one or

more "child" processing tasks during execution. The task/work unit **207** receives the tasks and dynamically schedules the processing tasks and child processing tasks for execution by the GPCs **208**.

FIG. **3**A is a block diagram of the task/work unit **207** of FIG. **2**, according to one embodiment of the present disclosure. The task/work unit **207** includes a task management unit **300** and the work distribution unit **340**. The task management unit **300** organizes tasks to be scheduled based on execution priority levels. For each priority level, the task management unit **300** stores a list of pointers to the TMDs **322** corresponding to the tasks in the scheduler table **321**, where the list may be implemented as a linked list. The TMDs **322** may be stored in the PP memory **204** or system memory **104**. The rate at which the task management unit **300** accepts tasks and stores the tasks in the scheduler table **321** is decoupled from the rate at which the task management unit **300** schedules tasks for execution. Therefore, the task management unit **300** may collect several tasks before scheduling the tasks. The collected tasks may then be scheduled based on priority information or using other techniques, such as round-robin scheduling.

The work distribution unit **340** includes a task table **345** with slots that may each be occupied by the TMD **322** for a task that is being executed. The task management unit **300** may schedule tasks for execution when there is a free slot in the task table **345**. When there is not a free slot, a higher priority task that does not occupy a slot may evict a lower priority task that does occupy a slot. When a task is evicted, the task is stopped, and if execution of the task is not complete, then a pointer to the task is added to a list of task pointers to be scheduled so that execution of the task will resume at a later time. When a child processing task is generated, during execution of a task, a pointer to the child task is added to the list of task pointers to be scheduled. A child task may be generated by a TMD **322** executing in the processing cluster array **230**.

Unlike a task that is received by the task/work unit **207** from the front end **212**, child tasks are received from the processing cluster array **230**. Child tasks are not inserted into pushbuffers or transmitted to the front end. The CPU **102** is not notified when a child task is generated or data for the child task is stored in memory. Another difference between the tasks that are provided through pushbuffers and child tasks is that the tasks provided through the pushbuffers are defined by the application program whereas the child tasks are dynamically generated during execution of the tasks.

Task Processing Overview

FIG. **3**B is a block diagram of a GPC **208** within one of the PPUs **202** of FIG. **2**, according to one embodiment of the present disclosure. Each GPC **208** may be configured to execute a large number of threads in parallel, where the term "thread" refers to an instance of a particular program executing on a particular set of input data. In some embodiments, single-instruction, multiple-data (SIMD) instruction issue techniques are used to support parallel execution of a large number of threads without providing multiple independent instruction units. In other embodiments, single-instruction, multiple-thread (SIMT) techniques are used to support parallel execution of a large number of generally synchronized threads, using a common instruction unit configured to issue instructions to a set of processing engines within each one of the GPCs **208**. Unlike a SIMD execution regime, where all processing engines typically execute identical instructions, SIMT execution allows different threads to more readily follow divergent execution paths through a given thread program. Persons of ordinary skill in the art will understand that a SIMD processing regime represents a functional subset of a SIMT processing regime.

Operation of GPC **208** is advantageously controlled via a pipeline manager **305** that distributes processing tasks to streaming multiprocessors (SMs) **310**. Pipeline manager **305** may also be configured to control a work distribution crossbar **330** by specifying destinations for processed data output by SMs **310**.

In one embodiment, each GPC **208** includes a number M of SMs **310**, where M≥1, each SM **310** configured to process one or more thread groups. Also, each SM **310** advantageously includes an identical set of functional execution units (e.g., execution units and load-store units—shown as Exec units **302** and LSUs **303** in FIG. **3**C) that may be pipelined, allowing a new instruction to be issued before a previous instruction has finished, as is known in the art. Any combination of functional execution units may be provided. In one embodiment, the functional units support a variety of operations including integer and floating point arithmetic (e.g., addition and multiplication), comparison operations, Boolean operations (AND, OR, XOR), bit-shifting, and computation of various algebraic functions (e.g., planar interpolation, trigonometric, exponential, and logarithmic functions, etc.); and the same functional unit hardware can be leveraged to perform different operations.

The series of instructions transmitted to a particular GPC **208** constitutes a thread, as previously defined herein, and the collection of a certain number of concurrently executing threads across the parallel processing engines (not shown) within an SM **310** is referred to herein as a "warp" or "thread group." As used herein, a "thread group" refers to a group of threads concurrently executing the same program on different input data, with one thread of the group being assigned to a different processing engine within an SM **310**. A thread group may include fewer threads than the number of processing engines within the SM **310**, in which case some processing engines will be idle during cycles when that thread group is being processed. A thread group may also include more threads than the number of processing engines within the SM **310**, in which case processing will take place over consecutive clock cycles. Since each SM **310** can support up to G thread groups concurrently, it follows that up to G*M thread groups can be executing in GPC **208** at any given time.

Additionally, a plurality of related thread groups may be active (in different phases of execution) at the same time within an SM **310**. This collection of thread groups is referred to herein as a "cooperative thread array" ("CTA") or "thread array." The size of a particular CTA is equal to m*k, where k is the number of concurrently executing threads in a thread group and is typically an integer multiple of the number of parallel processing engines within the SM **310**, and m is the number of thread groups simultaneously active within the SM **310**. The size of a CTA is generally determined by the programmer and the amount of hardware resources, such as memory or registers, available to the CTA.

Each SM **310** contains a level one (L1) cache (shown in FIG. **3**C) or uses space in a corresponding L1 cache outside of the SM **310** that is used to perform load and store operations. Each SM **310** also has access to level two (L2) caches that are shared among all GPCs **208** and may be used to transfer data between threads. Finally, SMs **310** also have access to off-chip "global" memory, which can include, e.g., parallel processing memory **204** and/or system memory **104**. It is to be understood that any memory external to PPU **202** may be used as global memory. Additionally, a level one-point-five

(L1.5) cache **335** may be included within the GPC **208**, configured to receive and hold data fetched from memory via memory interface **214** requested by SM **310**, including instructions, uniform data, and constant data, and provide the requested data to SM **310**. Embodiments having multiple SMs **310** in GPC **208** beneficially share common instructions and data cached in L1.5 cache **335**.

Each GPC **208** may include a memory management unit (MMU) **328** that is configured to map virtual addresses into physical addresses. In other embodiments, MMU(s) **328** may reside within the memory interface **214**. The MMU **328** includes a set of page table entries (PTEs) used to map a virtual address to a physical address of a tile and optionally a cache line index. The MMU **328** may include address translation lookaside buffers (TLB) or caches which may reside within multiprocessor SM **310** or the L1 cache or GPC **208**. The physical address is processed to distribute surface data access locality to allow efficient request interleaving among partition units **215**. The cache line index may be used to determine whether or not a request for a cache line is a hit or miss.

In graphics and computing applications, a GPC **208** may be configured such that each SM **310** is coupled to a texture unit **315** for performing texture mapping operations, e.g., determining texture sample positions, reading texture data, and filtering the texture data. Texture data is read from an internal texture L1 cache (not shown) or in some embodiments from the L1 cache within SM **310** and is fetched from an L2 cache that is shared between all GPCs **208**, parallel processing memory **204**, or system memory **104**, as needed. Each SM **310** outputs processed tasks to work distribution crossbar **330** in order to provide the processed task to another GPC **208** for further processing or to store the processed task in an L2 cache, parallel processing memory **204**, or system memory **104** via crossbar unit **210**. A preROP (pre-raster operations) **325** is configured to receive data from SM **310**, direct data to ROP units within partition units **215**, and perform optimizations for color blending, organize pixel color data, and perform address translations.

It will be appreciated that the core architecture described herein is illustrative and that variations and modifications are possible. Any number of processing units, e.g., SMs **310** or texture units **315**, preROPs **325** may be included within a GPC **208**. Further, as shown in FIG. **2**, a PPU **202** may include any number of GPCs **208** that are advantageously functionally similar to one another so that execution behavior does not depend on which GPC **208** receives a particular processing task. Further, each GPC **208** advantageously operates independently of other GPCs **208** using separate and distinct processing units, L1 caches to execute tasks for one or more application programs.

Persons of ordinary skill in the art will understand that the architecture described in FIGS. **1**, **2**, **3**A, and **3**B in no way limits the scope of the present invention and that the techniques taught herein may be implemented on any properly configured processing unit, including, without limitation, one or more CPUs, one or more multi-core CPUs, one or more PPUs **202**, one or more GPCs **208**, one or more graphics or special purpose processing units, or the like, without departing the scope of the present invention.

In embodiments of the present invention, it is desirable to use PPU **202** or other processor(s) of a computing system to execute general-purpose computations using thread arrays. Each thread in the thread array is assigned a unique thread identifier ("thread ID") that is accessible to the thread during the thread's execution. The thread ID, which can be defined as a one-dimensional or multi-dimensional numerical value controls various aspects of the thread's processing behavior. For instance, a thread ID may be used to determine which portion of the input data set a thread is to process and/or to determine which portion of an output data set a thread is to produce or write.

A sequence of per-thread instructions may include at least one instruction that defines a cooperative behavior between the representative thread and one or more other threads of the thread array. For example, the sequence of per-thread instructions might include an instruction to suspend execution of operations for the representative thread at a particular point in the sequence until such time as one or more of the other threads reach that particular point, an instruction for the representative thread to store data in a shared memory to which one or more of the other threads have access, an instruction for the representative thread to atomically read and update data stored in a shared memory to which one or more of the other threads have access based on their thread IDs, or the like. The CTA program can also include an instruction to compute an address in the shared memory from which data is to be read, with the address being a function of thread ID. By defining suitable functions and providing synchronization techniques, data can be written to a given location in shared memory by one thread of a CTA and read from that location by a different thread of the same CTA in a predictable manner. Consequently, any desired pattern of data sharing among threads can be supported, and any thread in a CTA can share data with any other thread in the same CTA. The extent, if any, of data sharing among threads of a CTA is determined by the CTA program; thus, it is to be understood that in a particular application that uses CTAs, the threads of a CTA might or might not actually share data with each other, depending on the CTA program, and the terms "CTA" and "thread array" are used synonymously herein.

FIG. **3**C is a block diagram of the SM **310** of FIG. **3**B, according to one embodiment of the present disclosure. The SM **310** includes an instruction L1 cache **370** that is configured to receive instructions and constants from memory via L1.5 cache **335**. A warp scheduler and instruction unit **312** receives instructions and constants from the instruction L1 cache **370** and controls local register file **304** and SM **310** functional units according to the instructions and constants. The SM **310** functional units include N exec (execution or processing) units **302** and P load-store units (LSU) **303**.

SM **310** provides on-chip (internal) data storage with different levels of accessibility. Special registers (not shown) are readable but not writeable by LSU **303** and are used to store parameters defining each thread's "position." In one embodiment, special registers include one register per thread (or per exec unit **302** within SM **310**) that stores a thread ID; each thread ID register is accessible only by a respective one of the exec unit **302**. Special registers may also include additional registers, readable by all threads that execute the same processing task represented by a TMD **322** (or by all LSUs **303**) that store a CTA identifier, the CTA dimensions, the dimensions of a grid to which the CTA belongs (or queue position if the TMD **322** encodes a queue task instead of a grid task), and an identifier of the TMD **322** to which the CTA is assigned.

If the TMD **322** is a grid TMD, execution of the TMD **322** causes a fixed number of CTAs to be launched and executed to process the fixed amount of data stored in the queue **525**. The number of CTAs is specified as the product of the grid width, height, and depth. The fixed amount of data may be stored in the TMD **322** or the TMD **322** may store a pointer to the data that will be processed by the CTAs. The TMD **322** also stores a starting address of the program that is executed by the CTAs.

If the TMD **322** is a queue TMD, then a queue feature of the TMD **322** is used, meaning that the amount of data to be processed is not necessarily fixed. Queue entries store data for processing by the CTAs assigned to the TMD **322**. The queue entries may also represent a child task that is generated by another TMD **322** during execution of a thread, thereby providing nested parallelism. Typically, execution of the thread, or CTA that includes the thread, is suspended until execution of the child task completes. The queue may be stored in the TMD **322** or separately from the TMD **322**, in which case the TMD **322** stores a queue pointer to the queue. Advantageously, data generated by the child task may be written to the queue while the TMD **322** representing the child task is executing. The queue may be implemented as a circular queue so that the total amount of data is not limited to the size of the queue.

CTAs that belong to a grid have implicit grid width, height, and depth parameters indicating the position of the respective CTA within the grid. Special registers are written during initialization in response to commands received via front end **212** from device driver **103** and do not change during execution of a processing task. The front end **212** schedules each processing task for execution. Each CTA is associated with a specific TMD **322** for concurrent execution of one or more tasks. Additionally, a single GPC **208** may execute multiple tasks concurrently.

A parameter memory (not shown) stores runtime parameters (constants) that can be read but not written by any thread within the same CTA (or any LSU **303**). In one embodiment, device driver **103** provides parameters to the parameter memory before directing SM **310** to begin execution of a task that uses these parameters. Any thread within any CTA (or any exec unit **302** within SM **310**) can access global memory through a memory interface **214**. Portions of global memory may be stored in the L1 cache **320**.

Local register file **304** is used by each thread as scratch space; each register is allocated for the exclusive use of one thread, and data in any of local register file **304** is accessible only to the thread to which the register is allocated. Local register file **304** can be implemented as a register file that is physically or logically divided into P lanes, each having some number of entries (where each entry might store, e.g., a 32-bit word). One lane is assigned to each of the N exec units **302** and P load-store units LSU **303**, and corresponding entries in different lanes can be populated with data for different threads executing the same program to facilitate SIMD execution. Different portions of the lanes can be allocated to different ones of the G concurrent thread groups, so that a given entry in the local register file **304** is accessible only to a particular thread. In one embodiment, certain entries within the local register file **304** are reserved for storing thread identifiers, implementing one of the special registers. Additionally, a uniform L1 cache **375** stores uniform or constant values for each lane of the N exec units **302** and P load-store units LSU **303**.

Shared memory **306** is accessible to threads within a single CTA; in other words, any location in shared memory **306** is accessible to any thread within the same CTA (or to any processing engine within SM **310**). Shared memory **306** can be implemented as a shared register file or shared on-chip cache memory with an interconnect that allows any processing engine to read from or write to any location in the shared memory. In other embodiments, shared state space might map onto a per-CTA region of off-chip memory, and be cached in L1 cache **320**. The parameter memory can be implemented as a designated section within the same shared register file or shared cache memory that implements shared memory **306**, or as a separate shared register file or on-chip cache memory to which the LSUs **303** have read-only access. In one embodiment, the area that implements the parameter memory is also used to store the CTA ID and task ID, as well as CTA and grid dimensions or queue position, implementing portions of the special registers. Each LSU **303** in SM **310** is coupled to a unified address mapping unit **352** that converts an address provided for load and store instructions that are specified in a unified memory space into an address in each distinct memory space. Consequently, an instruction may be used to access any of the local, shared, or global memory spaces by specifying an address in the unified memory space.

The L1 cache **320** in each SM **310** can be used to cache private per-thread local data and also per-application global data. In some embodiments, the per-CTA shared data may be cached in the L1 cache **320**. The LSUs **303** are coupled to the shared memory **306** and the L1 cache **320** via a memory and cache interconnect **380**.

Operand Caching

FIG. **4** is a block diagram of the SM **310** of FIG. **3**B, according to another example embodiment of the present disclosure. Although not shown explicitly, SM **310** of FIG. **4** may contain some or all of the components of SM **310** of FIG. **3**C, described above, in addition to the components explicitly shown in FIG. **4**. As shown in FIG. **4**, SM **310** includes the warp scheduler and instruction unit **312**, the local register file **304**, and one or more functional executive units, such as execution unit **302** or LSU **303**. Warp scheduler and instruction unit **312** includes a decode unit **450** and a dispatch unit **470**. Decode unit **450** receives the next instruction to be dispatched to execution unit **302**. The decode unit **450** performs a full decode of the instruction and transmits the decoded instruction to the dispatch unit **470**. For example, decode unit **450** will determine the particular type of instruction specified by the opcode in the instruction and the particular register indices that are specified as operands to the instruction as well as a register index for storing the result of an instruction. In some embodiments, instructions may be dual or quad issued and decode unit **450** may implement separate decode logic for each issued instruction. Dispatch unit **470** implements a FIFO and writes the decoded values to local register file **304** for execution. In embodiments that issue multiple instructions simultaneously, dispatch unit **470** may issue each instruction to a different portion of the functional units of SM **310**.

In one embodiment, local register file **304** includes four banks of registers (bank_0 **422**, bank_1 **424**, bank_2 **426**, and bank_3 **428**). In most conventional processing units, the local register file may be quite small. For example, the x86 CPU architecture includes 8 (32-bit) or 16 (64-bit) registers per processor core. In contrast, each bank of local register file **304** may include a large number of registers, such as 256 registers or more, available to use as inputs to the execution units **302**. Crossbar **420** is configured to connect the various registers in each of the register banks to an operand collector **440** in an execution unit **302**. Execution unit **302** implements a datapath for performing an operation. The datapath includes the operand collector **440**, the arithmetic logic unit (ALU) **452**, and a result FIFO **454**. Operand collector **440** includes a number of storage elements **441-446** that may be coupled to the inputs of the ALU **452**. Each storage element **441-446** may be a flip-flop, latch, or any other technically feasible circuit component capable of temporarily storing a value to supply as an input to the ALU **452**. The outputs of the storage elements may be hardwired to the circuit elements that com-

prise the ALU **452** such as an adder circuit or a floating point multiplier circuit. As shown, operand collector **440** includes six storage elements **441-446**. In other embodiments, any number of storage elements **441-446** may be implemented in execution unit **302**.

In conventional processor designs that include relatively small register files, a crossbar or other interconnect may be configured to couple any register in a local register file to any one of the datapath inputs during a single clock cycle. Furthermore, in conventional processors, a number of registers, the number equal to the number of inputs to the datapath, may simultaneously be connected to any of the datapath inputs in a single clock cycle. It will be appreciated that increasing the size of the local register file **304** beyond 8 or 16 registers may increase the complexity of crossbar **420**. Thus, in some embodiments, crossbar **420** may only couple a single register from each register bank (e.g., **422**, **424**, **426**, and **428**) to a particular storage element **441-446** during each clock cycle.

The processor architecture described above (where crossbar **420** couples only a single register per register bank to the operand collector **440** during one clock cycle) reduces the size and complexity of crossbar **420**, but the processor architecture also introduces various constraints for executing instructions via execution unit **302**. For example, the bandwidth between the local register file **304** and the operand collector **440** during each clock cycle is equal to the number of register banks in local register file **304**. As shown, local register file **304** includes four register banks, thereby allowing up to four operands to be written into operand collector **440** during a single clock cycle. However, because operand collector **440** includes six storage elements **441-446** for providing inputs to ALU **452** during each instruction, the operands may need to be loaded in two consecutive clock cycles (4 operands in the first clock cycle and 2 operands in the second clock cycle). In addition, care must be taken to avoid register bank conflicts by storing operand values needed for the same instruction in different register banks **422**, **424**, **426**, and **428**. For example, if all six operands are stored in bank_0 **422**, then six clock cycles are needed to load the operands into operand collector **440**. In one embodiment, dispatch unit **470** includes logic that is configured to avoid bank conflicts by ensuring that operands for each instruction are stored in different register banks **422**, **424**, **426**, and **428**. In another embodiment, driver **103** may include implement instructions that cause driver **103** to check and avoid bank conflicts during compilation of the parallel processor instructions before they are passed to PPU **202**. However, even with the implementation of such techniques, some bank conflicts may not be easily resolved.

In one embodiment, operand collector **440** enables the caching of operands associated with multiple instructions simultaneously within operand collector **440**. As shown, operand collector **440** includes one or more storage elements (**441(0)-441**(R−1); **442(0)-442**(R−1); etc.) corresponding to each input to ALU **452**. Dispatch unit **470** may include logic to control which sets of storage elements **441**(*i*)-**446**(*i*) to connect to ALU **452** for each instruction. For example, as shown in Table 1, a program executed on ALU **302** may include a set of instructions generated by driver **103** and transmitted to PPU **202**. A first instruction may be a fused multiply-add (FMA) instruction that includes three operands (R**13**, R**11**, R**14**) and an output (R**1**). A second instruction may be an addition (ADD) instruction that includes three operands (R**6**, R**7**, R**8**) and an output (R**2**). A third instruction may be a multiply (mUL) instruction that includes three operands (R**6**, R**11**, and R**13**). Operands indicated as X are not used as inputs to the ALU **452** for the given instruction and may be ignored for purposes of this disclosure.

TABLE 1

No.	Opcode	Output	Operands
1	FMA	R1	R13, R11, R14, X, X, X
2	ADD	R2	R6, R7, R8, X, X, X
3	MUL	R3	R6, R11, R13, X, X, X

Operand collector **440** may reuse an operand from the previous instruction during the next instruction. For example, during the second instruction above, the operand values stored in register **6**, register **7**, and register **8** in local register file **304** are loaded into storage elements **441**, **442**, and **443**, respectively. During the third instruction above, the value in storage element **441** may be reused, and only the operand values stored in register **11** and register **13** in local register file **304** are loaded into operand collector **440**. In other words, storage elements **441-446** do not need to be reloaded during every instruction. Dispatch unit **470** may include logic that stores the indices for each register currently loaded into the operand collector **440**. The logic included in dispatch unit **470** may be implemented as a lookup table (i.e., a cache table) that includes a slot corresponding to each storage element **441-446**. Each slot may store a value that indicates the register index associated with the value stored in the corresponding storage element **441-446**. When decode unit **450** determines which registers (i.e., register indices) to use as operands for the next instruction, dispatch unit **470** will check to determine whether the register value corresponding to that register index is currently stored in the same storage element **441-446** resulting from the scheduling of a previous instruction. Advantageously, if the same storage element **441-446** stores that register value, another register value from the same register bank may be loaded into a different storage element **441-446** during the same instruction cycle.

In another embodiment, operand collector **440** may store multiple sets of register values for a given datapath input of ALU **452** (i.e., operands **1-6** stored in storage elements **441-446**, respectively). In such embodiments, operand collector **440** may include a set of R storage elements **441(0)**, **441(1)**, . . . , **441**(R−1) for each datapath input of ALU **452**. Consequently, the operands from the previous R instructions may be cached in the operand collector **440** and reused in subsequent instructions to reduce the required bandwidth for loading operands from the local register file **304** into the operand collector **440**. For example, if R equals 2, then during the first instruction set forth above in Table 1, register **13** is loaded into storage element **441(0)**, register **11** is loaded into storage element **442(0)**, and register **14** is loaded into storage element **443(0)**. Subsequently, during the second instruction, register **6** is loaded into storage element **441(1)**, register **7** is loaded into storage element **442(1)**, and register **8** is loaded into storage element **443(1)**. During the third instruction, the first operand value (register **6**) is stored in storage element **441(1)** and the second operand value (register **11**) is stored in storage element **442(0)**.

In some embodiments, a given subset of storage elements **441**(*i*)-**446**(*i*) loaded during a given instruction must be selected concurrently for use with the subsequent instruction. In other words, during the third instruction discussed above, dispatch unit could reuse either the storage elements loaded during the first instruction **441(0)-446(0)** or the storage elements loaded during the second instruction **441(1)-446(1)**, but not the first storage element loaded during the second

instruction **441**(**1**) (i.e., storing the value in register **6**) and the second storage element loaded during the first instruction **442**(**0**) (i.e., storing the value in register **11**). In other embodiments, dispatch unit **470** may include logic for selecting between each individual storage element **441**-**446**, enabling register values stored during different instructions in different sets of storage elements **441**(*i*)-**446**(*i*) to be reused in a subsequent instruction.

It will be appreciated that cache coherency may be monitored to ensure that the values in the storage elements **441**-**446** are equal to the values in the corresponding registers in the local register file **304**. In one embodiment, the slots in the cache table store an index value that specifies the register corresponding to the value stored in the associated storage element **441**-**446**. Whenever a new value is written to a register in the local register file **304**, the cache table is searched to determine whether an index corresponding to that register is currently stored in the operand collector **440**. If the operand collector **440** includes an entry corresponding to that register, then the cache table is updated to remove that index from the slot (i.e., the value in that slot of the cache table is set to zero to invalidate the data in the operand collector **440**). Thus, if that register is then specified as an operand in a subsequent instruction, dispatch unit **470** will reload the value from the local register file **304** into the operand collector **440**, ensuring that invalid values are not provided to the ALU **452**.

FIG. **5** illustrates the operand collector **440** of FIG. **4**, according to one example embodiment of the present disclosure. As shown in FIG. **5**, operand collector **440** includes the storage elements **441**-**446** of FIG. **4**. As described above, the storage elements **441**-**446** implement a cache for storing register values coupled to the datapath inputs of ALU **452**. Crossbar **420** is configured by dispatch unit **470** to couple various registers in the local register file **304** to the storage elements **441**-**446**. Operand collector **440** also includes a plurality of multiplexors **511**-**516**. Multiplexor **511** has a first input coupled to a first storage element **441**(**0**) and a second input coupled to a second storage element **441**(**1**). Multiplexor **511** also has an output coupled to a first datapath input of ALU **452** that provides a first operand value to the logic of ALU **452**. Multiplexor **512** has a first input coupled to a first storage element **442**(**0**) and a second input coupled to a second storage element **442**(**1**). Multiplexor **512** also has an output coupled to a second datapath input of ALU **452** that provides a second operand value to the logic of ALU **452**. Multiplexor **513**, **514**, **515**, and **516** have similar connections to those described above with respect to multiplexors **511** and **512** for the third, fourth, fifth, and sixth datapath input of ALU **452**, respectively.

Dispatch unit **470** configures multiplexors **511**, **512**, **513**, **514**, **515**, and **516** to select between the different sets of storage elements **441**(*i*)-**446**(*i*). As shown in FIG. **5**, the control signal coupled to multiplexors **511**-**516** may be coupled to each of the multiplexors to select all of the storage elements **441**(*i*)-**446**(*i*) corresponding to the same previously loaded instruction. In alternative embodiments, dispatch unit **470** may supply a different control signal to each of the multiplexors **511**-**516**, allowing storage elements **441**(*i*)-**446**(*i*) corresponding to different instructions to be coupled to the datapath inputs of ALU **452**.

In alternative embodiments, operand collector **440** may include different interconnect logic that enables any of the storage elements **441**-**446** to connect to any of the datapath inputs of ALU **452**. For example, one or more crossbars may be implemented within operand collector **440** instead of multiplexors **511**-**516**. The one or more crossbars, as an example, may enable storage element **442**(**1**) to be connected with the fourth datapath input of ALU **452** and storage element **443**(**0**) to be connected with the first datapath input of ALU **452**. In other words, an operand will result in an operand cache **440** hit if the operand was previously loaded into any storage element of operand cache **440**. Thus, referring to the instructions given above in Table 1, all three operands specified by the third instruction may be reused because the three operands specified by the third instruction were previously loaded into at least one of the storage elements of operand cache **440** during the two previous instructions.

FIG. **6** is a flow chart illustrating a method **600** for executing instructions on a processor core implemented with source operand collector caching, according to one example embodiment of the present disclosure. Although the method steps are described in conjunction with the systems of FIGS. **1**, **2**, **3**A-**3**C, **4**, and **5** persons of ordinary skill in the art will understand that any system configured to perform the method steps, in any order, is within the scope of the disclosure.

Method **600** begins at step **610**, where SM **310** decodes an instruction to determine a plurality of operands specified by the instruction. Each operand may be specified as an index corresponding to a register within the local register file **304**. At step **612**, SM **310** determines whether one or more of the operands is cached in operand collector **440**. SM **310** may maintain a cache table that stores indices corresponding to the registers stored in operand collector **440**. Dispatch unit **470** may check the decoded register indexes corresponding to the operands specified by the instruction against the indexes stored in the cache table.

At step **614**, for each operand not stored in the operand collector **440**, SM **310** configures crossbar **420** to couple the specified registers in the local register file **304** with the storage elements **441**-**446** in the operand collector **440**. At step **616**, SM **310** updates the cache table to reflect the register indices corresponding to register values stored in operand collector **440**. At step **618**, SM **310** configures ALU **452** to execute the instruction using the operands stored in the operand collector **440** as inputs to ALU **452**. At step **620**, SM **310** determines whether there are additional instructions scheduled for execution on ALU **452**. If additional instructions are scheduled for execution, then method **600** returns to step **610** where the next instruction is decoded. However, if no more instructions are scheduled for execution, then method **600** terminates.

One embodiment of the disclosure may be implemented as a program product for use with a computer system. The program(s) of the program product define functions of the embodiments (including the methods described herein) and can be contained on a variety of computer-readable storage media. Illustrative computer-readable storage media include, but are not limited to: (i) non-writable storage media (e.g., read-only memory devices within a computer such as compact disc read only memory (CD-ROM) disks readable by a CD-ROM drive, flash memory, read only memory (ROM) chips or any type of solid-state non-volatile semiconductor memory) on which information is permanently stored; and (ii) writable storage media (e.g., floppy disks within a diskette drive or hard-disk drive or any type of solid-state random-access semiconductor memory) on which alterable information is stored.

The disclosure has been described above with reference to specific embodiments. Persons of ordinary skill in the art, however, will understand that various modifications and changes may be made thereto without departing from the broader spirit and scope of the disclosure as set forth in the

appended claims. The foregoing description and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

What is claimed is:

1. A method for executing instructions on a processor core implemented with a source operand collector cache coupled to at least one input of a data path residing within the processor core, the method comprising:

decoding an instruction to determine a plurality of operands specified by the instruction;

for each operand in the plurality of operands, determining that the operand is not stored in the source operand collector cache, loading the operand into the source operand collector cache from a local register file; and

configuring the processor core to execute the instruction across the operands stored in the source operand collector cache.

2. The method of claim **1**, further comprising, upon loading the operand into the source operand collector cache, updating a cache table to associate a register index corresponding with the operand with a storage element in the source operand collector cache.

3. The method of claim **2**, further comprising:

determining that the processor core has written a value to a register in the local register file; and

in response to determining that the value has been written to the register, updating the cache table to remove any entries associated with the register.

4. The method of claim **1**, wherein the local register file comprises a plurality of register banks.

5. The method of claim **4**, further comprising configuring a crossbar to couple a register in a first register bank in the plurality of register banks to a first storage element in the source operand collector cache.

6. The method of claim **5**, wherein the crossbar is configured to couple only a single register from each register bank in the plurality of register banks to the source operand collector cache during each clock cycle.

7. The method of claim **1**, wherein the source operand collector cache is sized to concurrently store the plurality of operands specified by the instruction and a plurality of operands specified by another decoded instruction.

8. A parallel processing unit, comprising:

a processor core including a data path for executing instructions;

a source operand collector cache coupled to at least one input of the data path;

a local register file coupled to the source operand collector cache; and

a scheduling unit configured to:

decode an instruction to determine a plurality of operands specified by the instruction,

for each operand in the plurality of operands, determining that the operand is not stored in the source operand collector cache, load the operand into the source operand collector cache from the local register file, and

configure the processor core to execute the instruction across the operands stored in the source operand collector cache.

9. The parallel processing unit of claim **8**, further comprising a cache table, and wherein, upon loading the operand into the source operand collector cache, the scheduling unit is further configured to update the cache table to associate a register index corresponding with the operand with a storage element in the source operand collector cache.

10. The parallel processing unit of claim **8**, wherein the local register file comprises a plurality of register banks.

11. The parallel processing unit of claim **10**, further comprising a crossbar configured to couple a register in a first register bank in the plurality of register banks to a first storage element in the source operand collector cache.

12. The parallel processing unit of claim **11**, wherein the crossbar is configured to couple only a single register from each register bank in the plurality of register banks to the source operand collector cache during each clock cycle.

13. The parallel processing unit of claim **8**, wherein the source operand collector cache is sized to concurrently store the plurality of operands specified by the instruction and a plurality of operands specified by another decoded instruction.

14. A computer system, comprising:

a memory; and

a parallel processing unit coupled to the memory and including:

a processor core including a data path for executing instructions;

a source operand collector cache coupled to at least one input of the data path;

a local register file coupled to the source operand collector cache; and

a scheduling unit configured to:

decode an instruction to determine a plurality of operands specified by the instruction,

for each operand in the plurality of operands, determining that the operand is not stored in the source operand collector cache, load the operand into the source operand collector cache from the local register file, and

configure the processor core to execute the instruction across the operands stored in the source operand collector cache.

15. The computer system of claim **14**, further comprising a cache table, and wherein, upon loading the operand into the source operand collector cache, the scheduling unit is further configured to update the cache table to associate a register index corresponding with the operand with a storage element in the source operand collector cache.

16. The computer system of claim **14**, wherein the local register file comprises a plurality of register banks.

17. The computer system of claim **16**, further comprising a crossbar configured to couple a register in a first register bank in the plurality of register banks to a first storage element in the source operand collector cache.

18. The computer system of claim **17**, wherein the crossbar is configured to couple only a single register from each register bank in the plurality of register banks to the source operand collector cache during each clock cycle.

19. The computer system of claim **14**, wherein the source operand collector cache is sized to concurrently store the plurality of operands specified by the instruction and a plurality of operands specified by another decoded instruction.

* * * * *