US011315310B2

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

(10) **Patent No.: US 11,315,310 B2**
(45) **Date of Patent: Apr. 26, 2022**

(54) **RESERVOIR-BASED SPATIOTEMPORAL IMPORTANCE RESAMPLING UTILIZING A GLOBAL ILLUMINATION DATA STRUCTURE**

(71) Applicant: **NVIDIA Corporation**, Santa Clara, CA (US)

(72) Inventors: **Christopher Ryan Wyman**, Redmond, WA (US); **Morgan McGuire**, Williamstown, MA (US); **Peter Schuyler Shirley**, Salt Lake City, UT (US); **Aaron Eliot Lefohn**, Kirkland, WA (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 0 days.

(21) Appl. No.: **17/152,460**

(22) Filed: **Jan. 19, 2021**

(65) **Prior Publication Data**

US 2021/0287426 A1 Sep. 16, 2021

Related U.S. Application Data

(60) Provisional application No. 62/988,789, filed on Mar. 12, 2020.

(51) **Int. Cl.**
G06T 15/50 (2011.01)
G06T 15/06 (2011.01)

(52) **U.S. Cl.**
CPC ***G06T 15/506*** (2013.01); ***G06T 15/06*** (2013.01)

(58) **Field of Classification Search**
CPC G06T 15/506; G06T 15/06
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

2008/0150938 A1* 6/2008 Pantaleoni G06T 15/50 345/419
2009/0213118 A1* 8/2009 Ha G06T 15/06 345/421
2013/0120384 A1* 5/2013 Jarosz G06F 17/14 345/426
2014/0327673 A1* 11/2014 Sousa G06T 15/506 345/426
2016/0216107 A1* 7/2016 Barker G06T 7/73
2017/0263043 A1* 9/2017 Peterson G06T 15/00
2018/0046885 A1* 2/2018 Barker G06T 7/66
2020/0320684 A1* 10/2020 Wiemker G06T 7/62
(Continued)

OTHER PUBLICATIONS

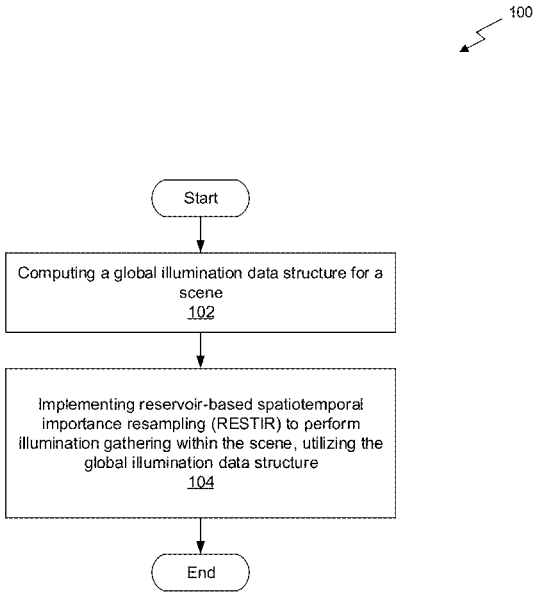
Greger et al., "The Irradiance Volume", 1998 (Year: 1998).*
(Continued)

Primary Examiner — Phong X Nguyen
(74) *Attorney, Agent, or Firm* — Zilka-Kotab, P.C.

(57) **ABSTRACT**

A global illumination data structure (e.g., a data structure created to store global illumination information for geometry within a scene to be rendered) is computed for the scene. Additionally, reservoir-based spatiotemporal importance resampling (RESTIR) is used to perform illumination gathering, utilizing the global illumination data structure. The illumination gathering includes identifying light values for points within the scene, where one or more points are selected within the scene based on the light values in order to perform ray tracing during the rendering of the scene.

22 Claims, 11 Drawing Sheets



(56) **References Cited**

U.S. PATENT DOCUMENTS

2020/0388004 A1* 12/2020 Zhang G06T 7/337
2021/0012246 A1* 1/2021 Hazard G06N 20/00

OTHER PUBLICATIONS

Talbot et al., "Importance Resampling for Global Illumination", 2005 (Year: 2005).*
Vitter, Random Sampling with a Reservoir, 1985 (Year: 1985).*
Efraimidis et al., Weighted random sampling with a reservoir, 2006 (Year: 2006).*
Huang et al., Sequentialized Sampling Importance Resampling and Scalable IWAE, 2017 (Year: 2017).*
Burke, NVIDIA RTX Technology Realizes Dream of Real-Time Cinematic Rendering, 2018 (Year: 2018).*
Majercik et al., "Scaling Probe-Based Real-Time Dynamic Global Illumination for Production," Journal of Computer Graphics Techniques, vol. 3, No. 1, 2014, pp. 1-28.
Bitterli et al., "Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting," ACM Transactions on Graphics, vol. 39, No. 4, Nov. 2020, 17 pages.

* cited by examiner

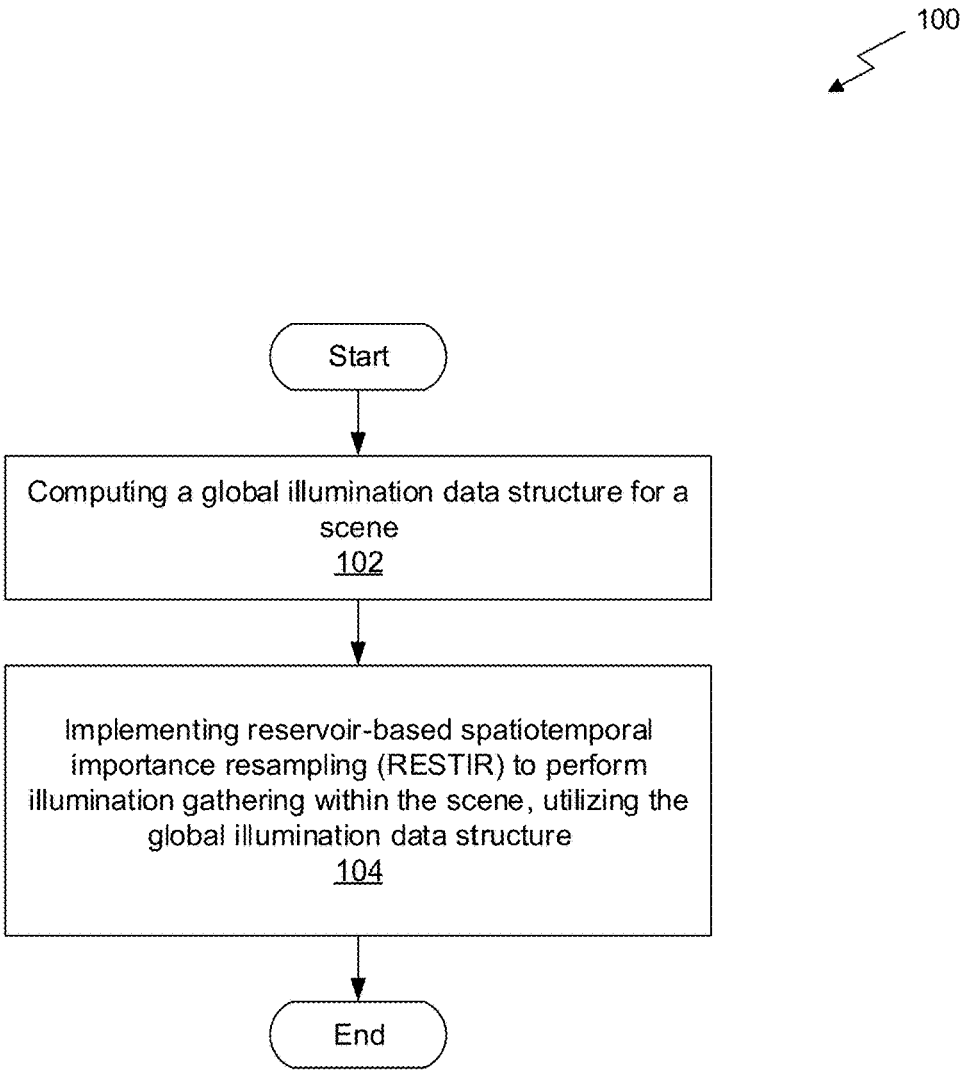

100
Start
Computing a global illumination data structure for a scene
102
Implementing reservoir-based spatiotemporal importance resampling (RESTIR) to perform illumination gathering within the scene, utilizing the global illumination data structure
104
End

Fig. 1

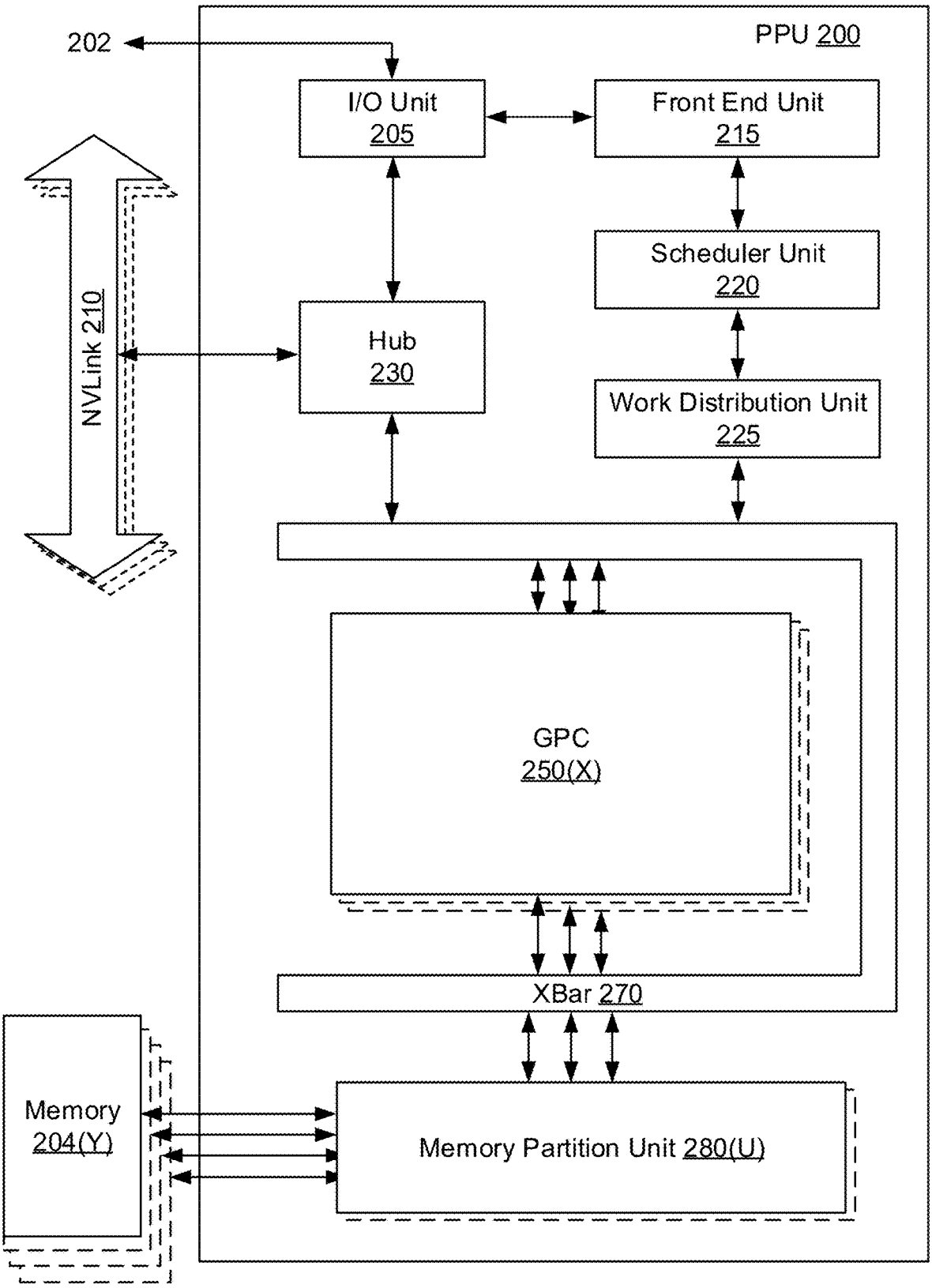

202
PPU 200
I/O Unit 205
Front End Unit 215
Scheduler Unit 220
Work Distribution Unit 225
Hub 230
NVLink 210
GPC 250(X)
XBar 270
Memory 204(Y)
Memory Partition Unit 280(U)

Fig. 2

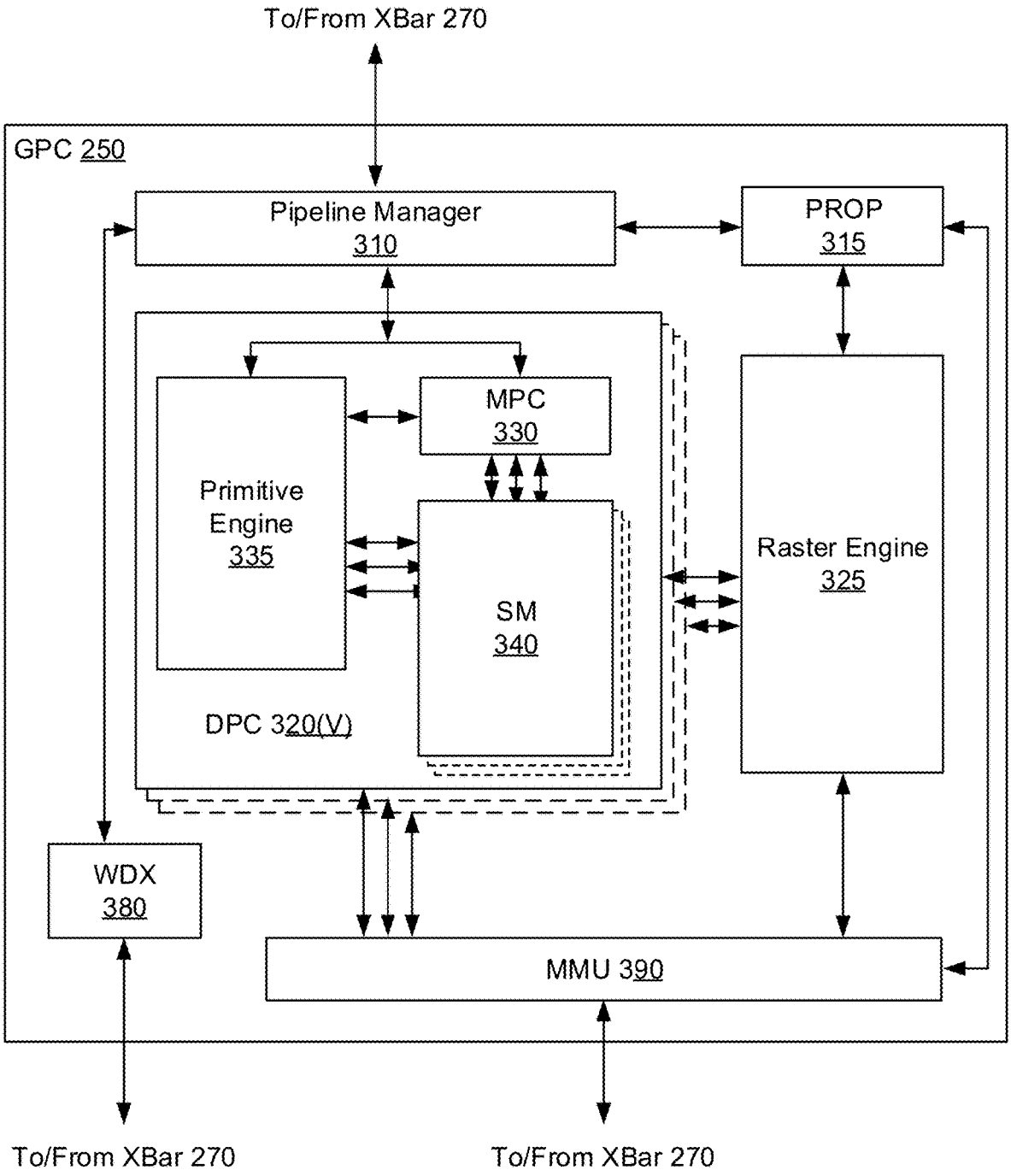

To/From XBar 270
GPC 250
Pipeline Manager
310
PROP
315
MPC
330
Primitive
Engine
335
SM
340
Raster Engine
325
DPC 320(V)
WDX
380
MMU 390
To/From XBar 270
To/From XBar 270

Fig. 3A

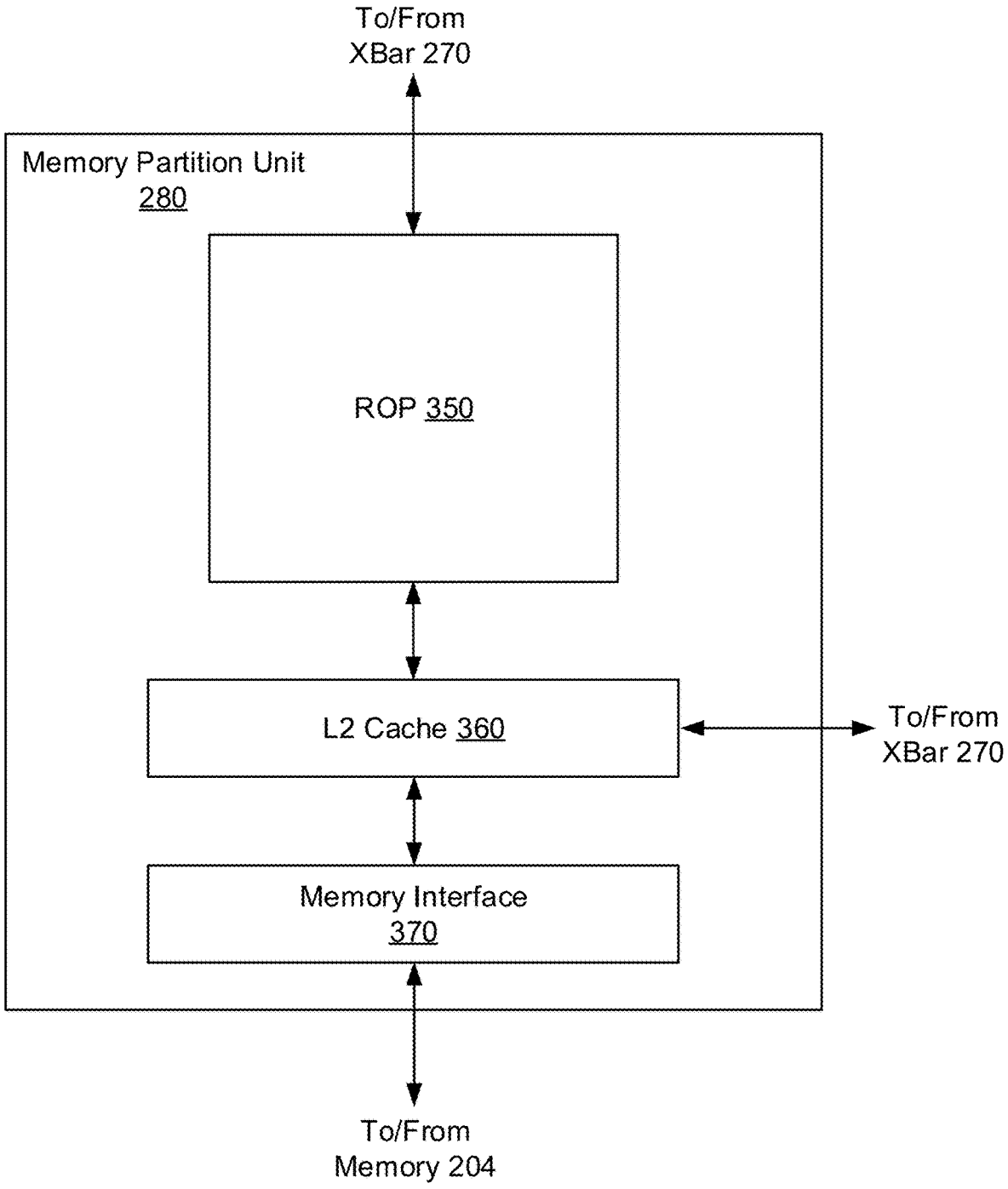

To/From
XBar 270
Memory Partition Unit
280
ROP 350
L2 Cache 360
To/From
XBar 270
Memory Interface
370
To/From
Memory 204

Fig. 3B

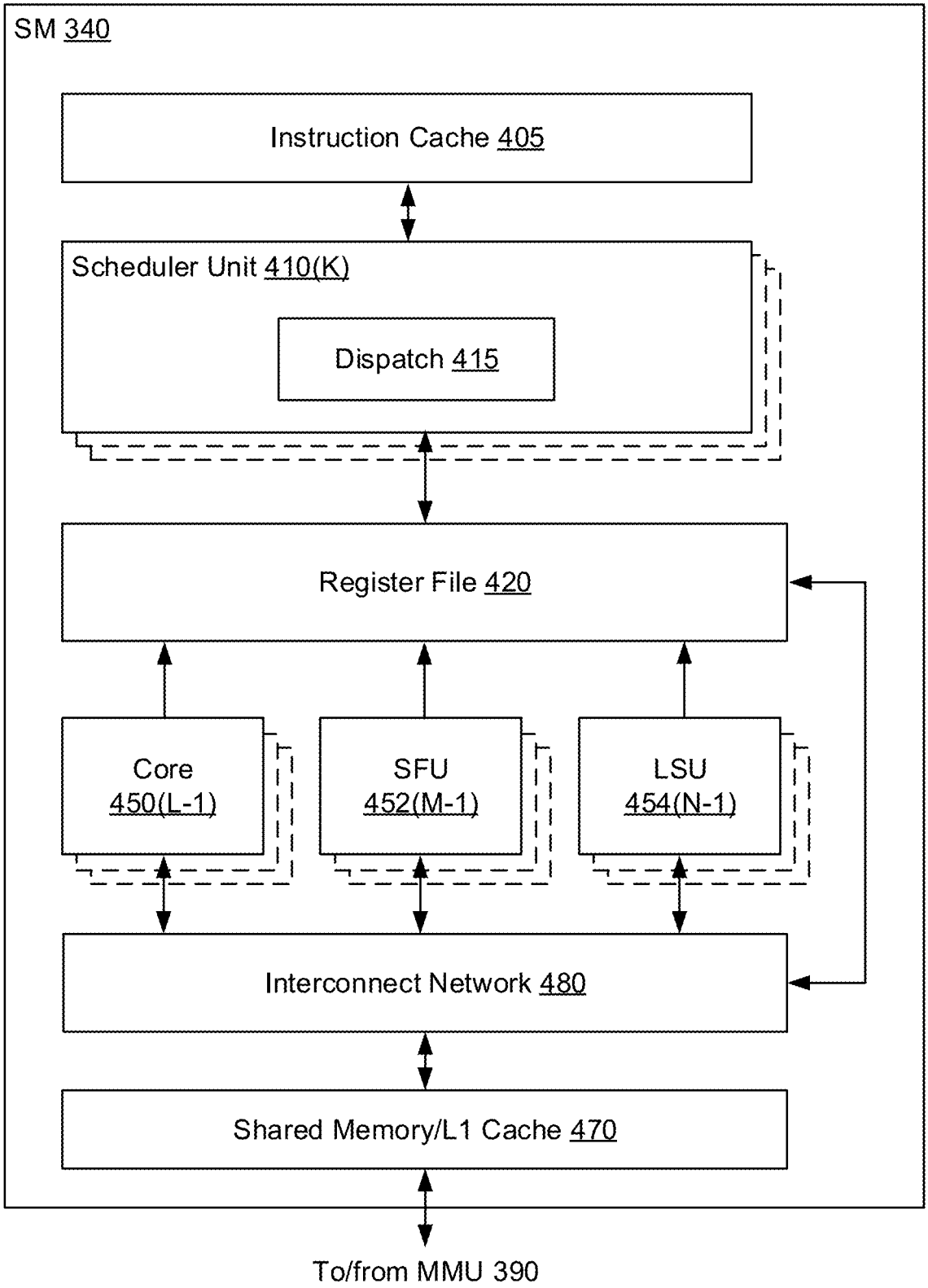

SM 340
Instruction Cache 405
Scheduler Unit 410(K)
Dispatch 415
Register File 420
Core 450(L-1)
SFU 452(M-1)
LSU 454(N-1)
Interconnect Network 480
Shared Memory/L1 Cache 470
To/from MMU 390

Fig. 4A

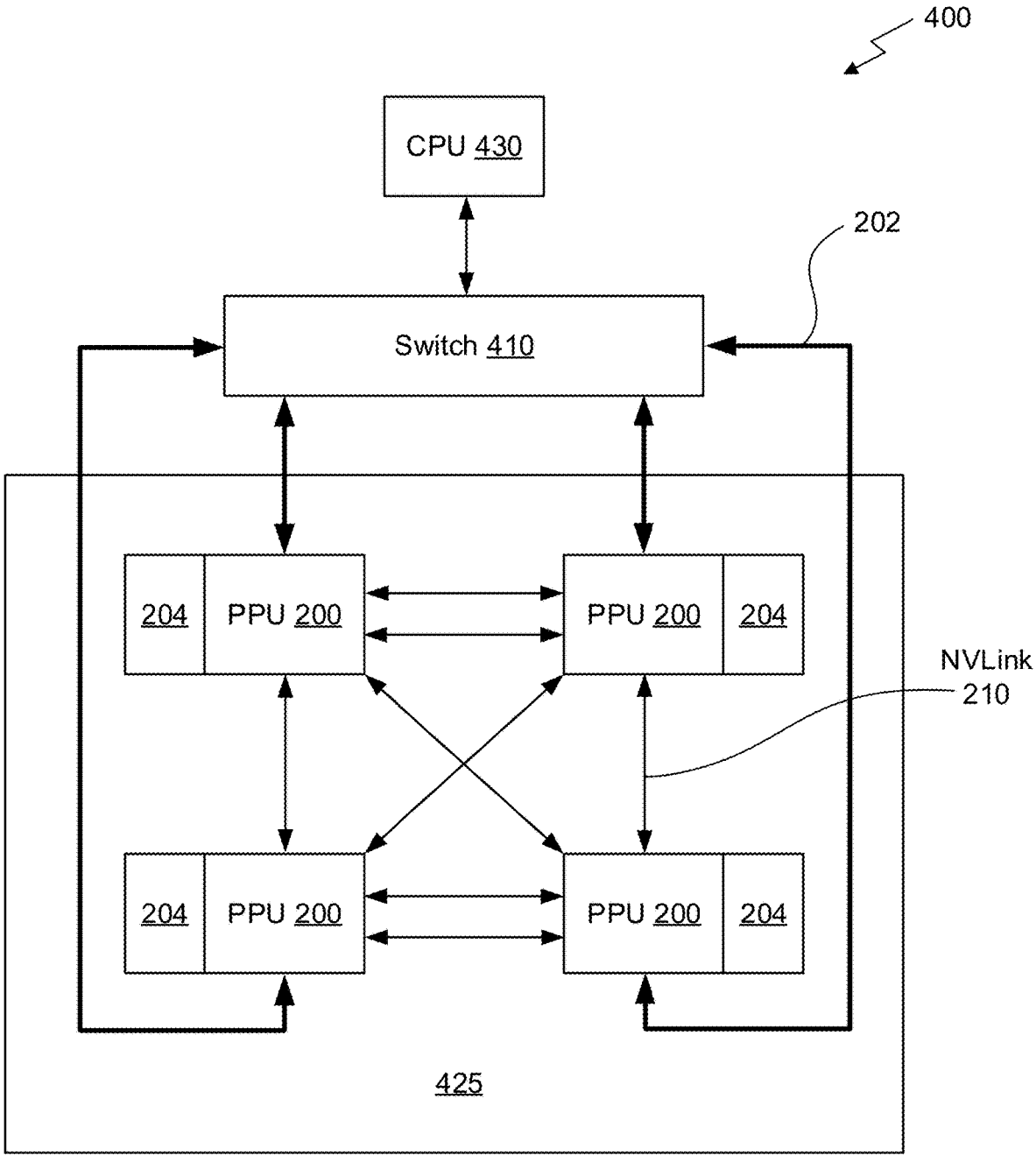

400
CPU 430
202
Switch 410
204
PPU 200
PPU 200
204
NVLink
210
204
PPU 200
PPU 200
204
425

Fig. 4B

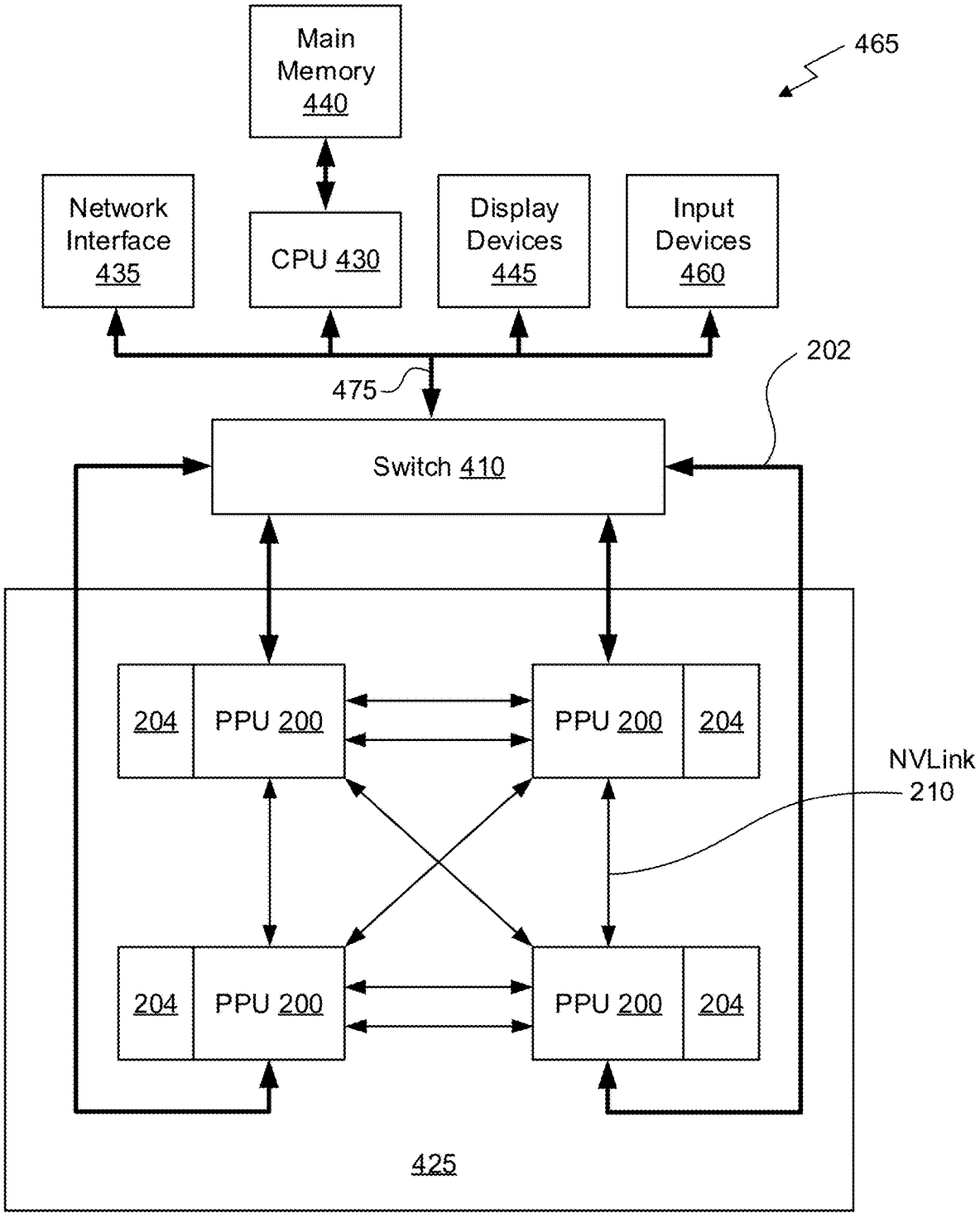

Main Memory 440
465
Network Interface 435
CPU 430
Display Devices 445
Input Devices 460
202
475
Switch 410
204
PPU 200
PPU 200
204
NVLink 210
204
PPU 200
PPU 200
204
425

Fig. 4C

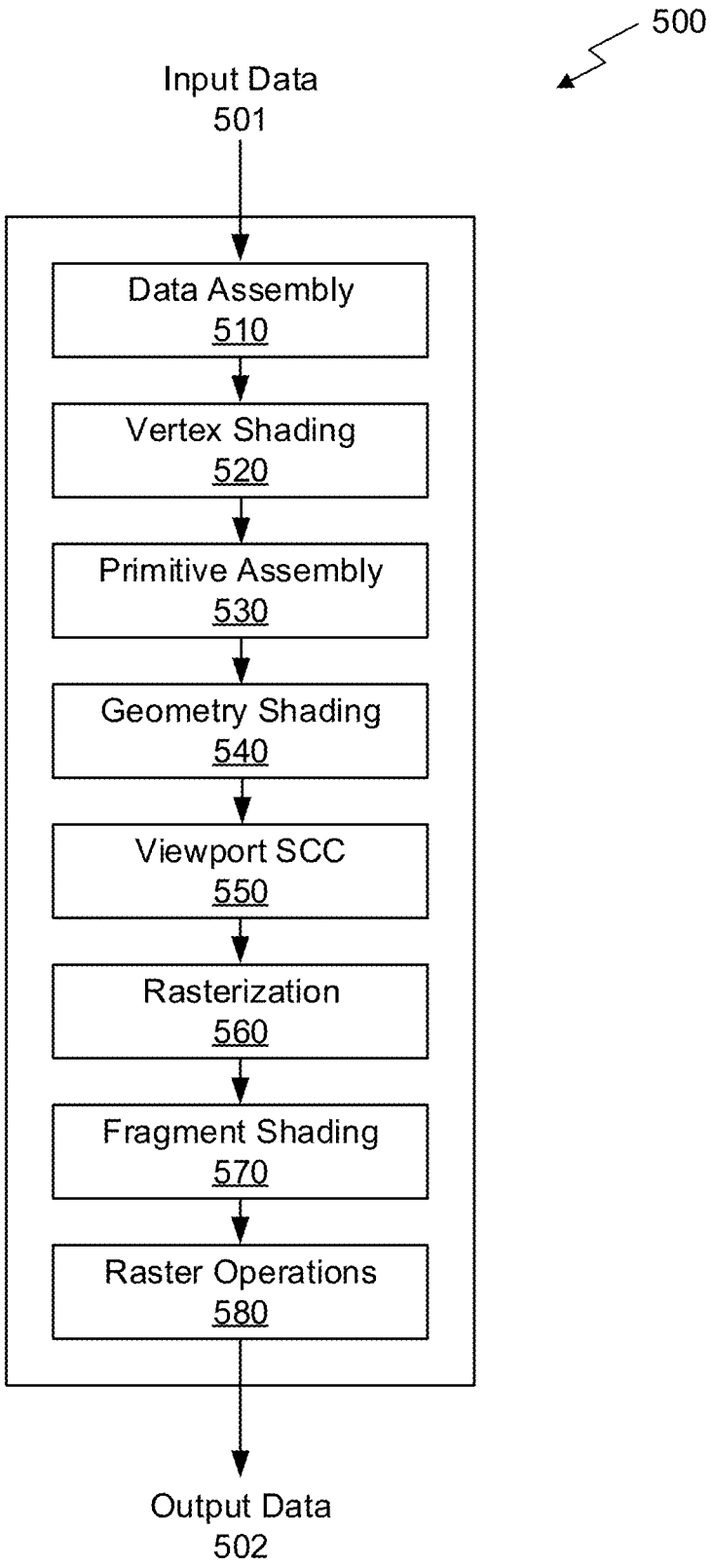

500
Input Data
501
Data Assembly
510
Vertex Shading
520
Primitive Assembly
530
Geometry Shading
540
Viewport SCC
550
Rasterization
560
Fragment Shading
570
Raster Operations
580
Output Data
502

Fig. 5

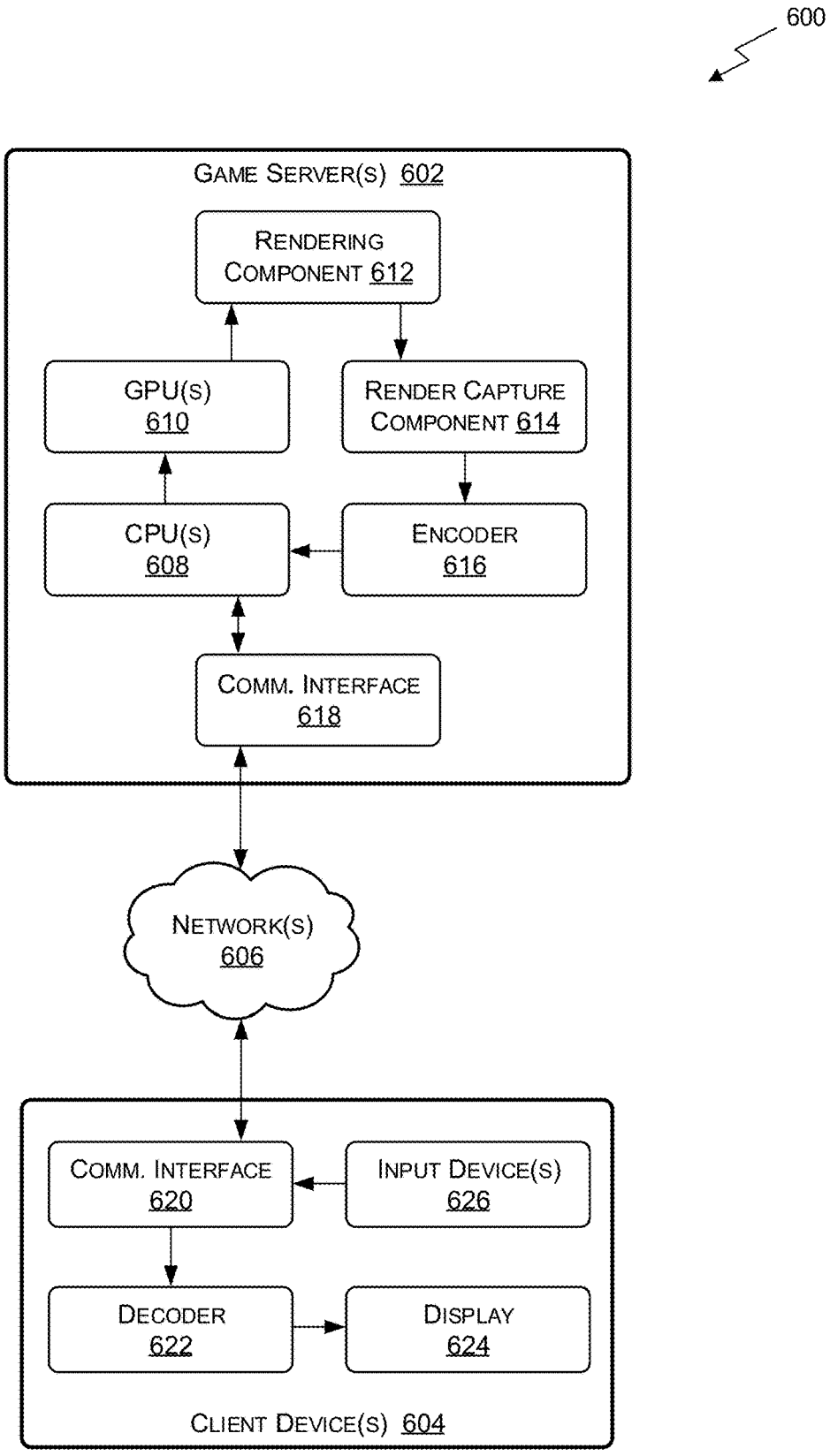

600
GAME SERVER(S) 602
RENDERING COMPONENT 612
GPU(S) 610
RENDER CAPTURE COMPONENT 614
CPU(S) 608
ENCODER 616
COMM. INTERFACE 618
NETWORK(S) 606
COMM. INTERFACE 620
INPUT DEVICE(S) 626
DECODER 622
DISPLAY 624
CLIENT DEVICE(S) 604

Fig. 6

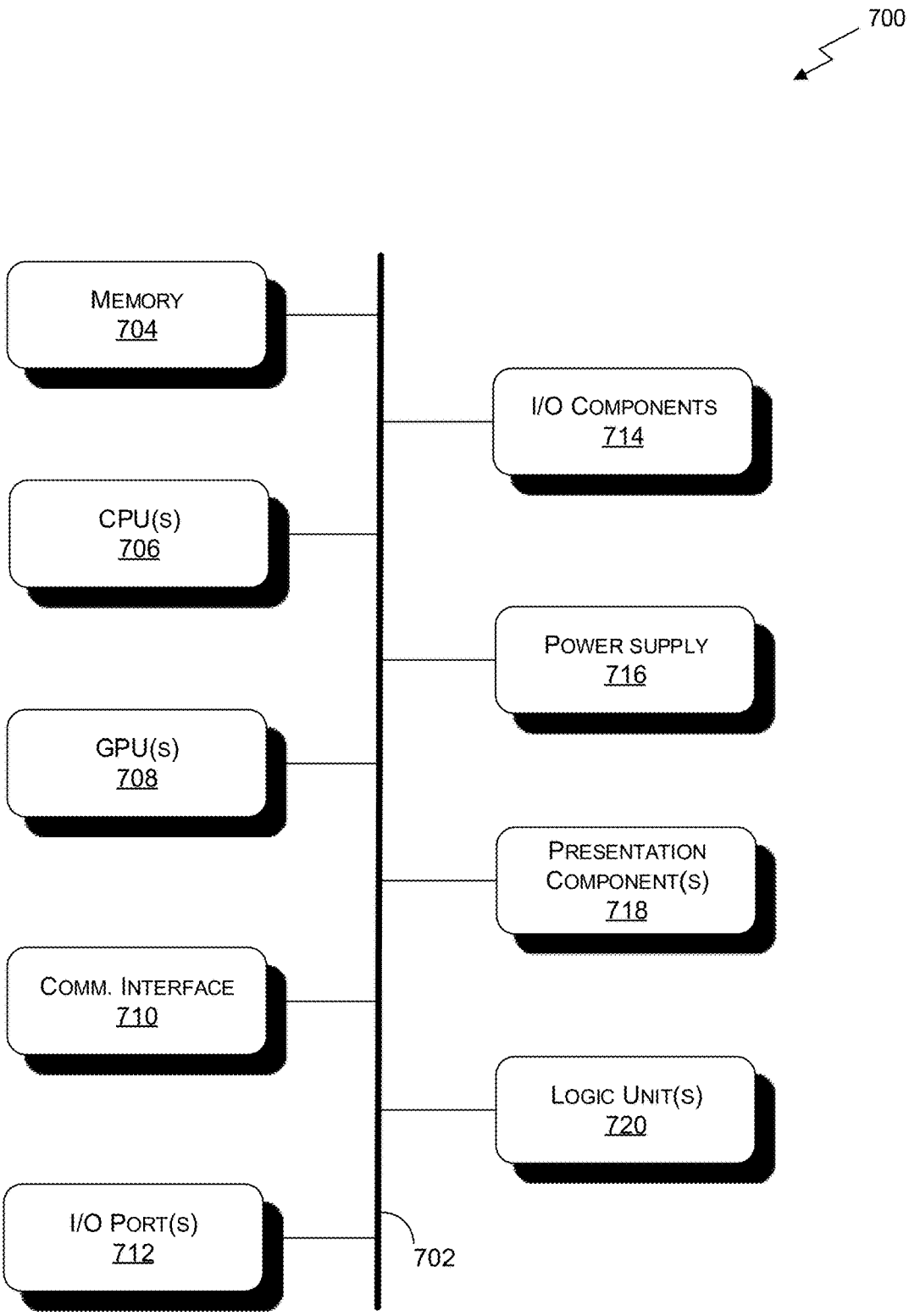

700
MEMORY
704
CPU(S)
706
GPU(S)
708
COMM. INTERFACE
710
I/O PORT(S)
712
I/O COMPONENTS
714
POWER SUPPLY
716
PRESENTATION
COMPONENT(S)
718
LOGIC UNIT(S)
720
702

Fig. 7

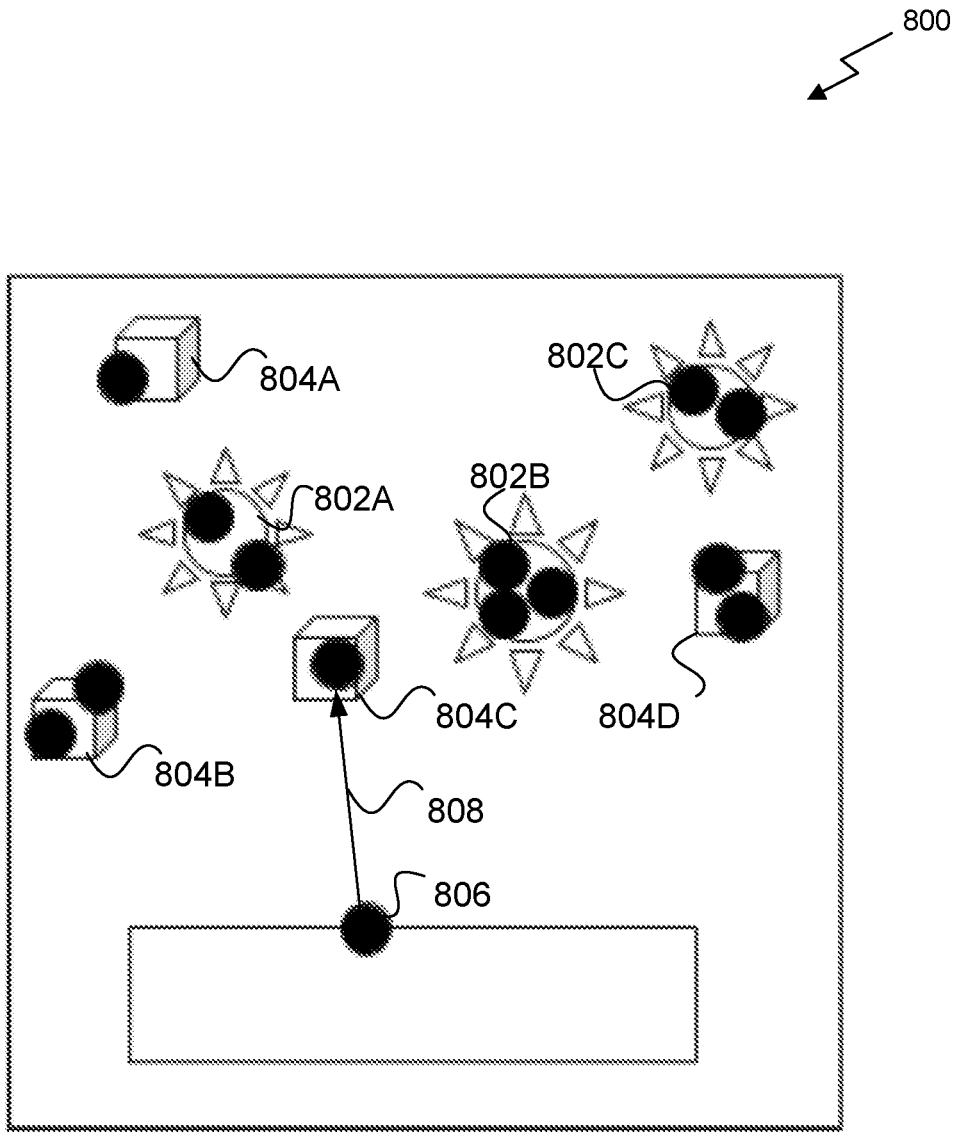


Fig. 8

RESERVOIR-BASED SPATIOTEMPORAL IMPORTANCE RESAMPLING UTILIZING A GLOBAL ILLUMINATION DATA STRUCTURE

CLAIM OF PRIORITY

This application claims the benefit of U.S. Provisional Application No. 62/988,789, filed on Mar. 12, 2020, which is hereby incorporated by reference in its entirety.

TECHNICAL FIELD

The present invention relates to image rendering, and more particularly to performing light gathering within a scene.

BACKGROUND

Reservoir importance resampling provides an effective means to compute lighting parameters from multiple light sources. However, current reservoir importance resampling implementations only consider direct lighting sources, and not indirect lighting. There is therefore a need to improve this resampling technique by including the consideration of indirect lighting during resampling.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** illustrates a flowchart of a method for reservoir-based spatiotemporal importance resampling utilizing a global illumination data structure, in accordance with an embodiment.

FIG. **2** illustrates a parallel processing unit, in accordance with an embodiment.

FIG. **3**A illustrates a general processing cluster within the parallel processing unit of FIG. **2**, in accordance with an embodiment.

FIG. **3**B illustrates a memory partition unit of the parallel processing unit of FIG. **2**, in accordance with an embodiment.

FIG. **4**A illustrates the streaming multi-processor of FIG. **3**A, in accordance with an embodiment.

FIG. **4**B is a conceptual diagram of a processing system implemented using the PPU of FIG. **2**, in accordance with an embodiment.

FIG. **4**C illustrates an exemplary system in which the various architecture and/or functionality of the various previous embodiments may be implemented.

FIG. **5** is a conceptual diagram of a graphics processing pipeline implemented by the PPU of FIG. **2**, in accordance with an embodiment.

FIG. **6** is a block diagram of an example game streaming system suitable for use in implementing some embodiments of the present disclosure.

FIG. **7** is a block diagram of an example computing device suitable for use in implementing some embodiments of the present disclosure.

FIG. **8** illustrates an exemplary scene for which illumination gathering is being performed utilizing RESTIR and a global illumination data structure, in accordance with one embodiment.

DETAILED DESCRIPTION

Current reservoir importance resampling is used during image rendering to determine lighting parameters from multiple light sources within a scene being rendered, but this method only considers direct lighting sources, and not indirect lighting. In response, a global illumination data structure (e.g., a data structure created to store global illumination information for geometry within a scene to be rendered) is computed for the scene. Additionally, reservoir-based spatiotemporal importance resampling (RESTIR) is used to perform illumination gathering, utilizing the global illumination data structure. The illumination gathering includes identifying light values for points within the scene, where one or more points are selected within the scene based on the light values in order to perform ray tracing during the rendering of the scene. In this way, indirect lighting may be considered as well as direct lighting during image rendering.

FIG. **1** illustrates a flowchart of a method **100** for reservoir-based spatiotemporal importance resampling utilizing a global illumination data structure, in accordance with an embodiment. Although method **100** is described in the context of a processing unit, the method **100** may also be performed by a program, custom circuitry, or by a combination of custom circuitry and a program. For example, the method **100** may be executed by a GPU (graphics processing unit), CPU (central processing unit), or any processor capable of performing parallel path space filtering by hashing. Furthermore, persons of ordinary skill in the art will understand that any system that performs method **100** is within the scope and spirit of embodiments of the present invention.

It should be understood that this and other arrangements described herein are set forth only as examples. Other arrangements and elements (e.g., machines, interfaces, functions, orders, groupings of functions, etc.) may be used in addition to or instead of those shown, and some elements may be omitted altogether. Further, many of the elements described herein are functional entities that may be implemented as discrete or distributed components or in conjunction with other components, and in any suitable combination and location. Various functions described herein as being performed by entities may be carried out by hardware, firmware, and/or software. For instance, various functions may be carried out by a processor executing instructions stored in memory.

As shown in operation **102**, a global illumination data structure is computed for a scene. In one embodiment, the global illumination data structure may include a world-space data structure such as one or more of an RTX global illumination (RTXGI) data structure, a photon map, a light map, a radiance cache, an irradiance cache, a radiosity value data structure, a light probe data structure, etc. In another embodiment, the global illumination information may include a sum of indirect and direct light values for a predetermined point/pixel within the scene.

Additionally, in one embodiment, the global illumination data structure may include any data structure created to store global illumination information for elements (e.g., surfaces (geometry), volumetrics (clouds), probe data, etc.) within the scene. For example, additional information (e.g., distance values from various points in the scene, direction information, and other heuristics information) may be stored within the global illumination data structure. In another example, the global illumination data structure may be hashed, implemented hierarchically, etc.

Also, as shown in operation **104**, reservoir-based spatiotemporal importance resampling (RESTIR) is implemented to perform illumination gathering within the scene, utilizing the global illumination data structure. In one embodiment, reservoir-based spatiotemporal importance

resampling (RESTIR) may identify a predetermined set of candidate light sources within a scene to be rendered.

Additionally, in one embodiment, RESTIR may identify a random set of points on the candidate light sources, and may evaluate their potential lighting contribution to the scene (e.g., their overall brightness, etc.). For example, the lighting contribution of each of the identified points may be determined utilizing a predetermined lighting equation. In another embodiment, using RESTIR, one or more of the random set of points on the candidate light sources may be selected, where a probability of point selection is proportional to the potential lighting contribution of the point.

Further, in one embodiment, a visibility ray (e.g., a shadow ray, etc.) may then be sent from a point being shaded in the scene to each of the selected points. In another embodiment, RESTIR may resample a set of candidate light samples and apply additional spatial and temporal resampling to leverage information from relevant nearby samples, utilizing a streaming reservoir algorithm.

Further still, in one embodiment, the global illumination data structure may expand RESTIR by treating all surfaces in the scene as candidate light sources during illumination gathering. For example, the surfaces may include all triangles (or other primitives/geometry) within the scene.

In addition, in one embodiment, the lighting contribution from actual candidate light sources may be determined utilizing a predetermined lighting equation within RESTIR. In another embodiment, the lighting contribution from all other surfaces within the scene may be determined by performing a data lookup within the global illumination data structure (e.g., instead of utilizing the predetermined lighting equation).

Furthermore, in one embodiment, both the lighting contribution from actual candidate light sources and the lighting contribution from all other surfaces within the scene may be determined utilizing the global illumination data structure. In this way, both direct and indirect lighting may be accounted for when implementing RESTIR within a scene to be rendered. More specifically, all surfaces within a scene to be rendered may be treated as candidate light sources (utilizing the global illumination data structure) when performing illumination gathering using RESTIR. This may improve ray tracing by reducing an amount of noise in a resulting rendered scene, increasing a level of detail in a resulting rendered scene, etc.

Further still, in one embodiment, results of the illumination gathering may be computed on a first computing device, a first processor, etc. and may be shared with another separate computing device, another processor, etc. In another embodiment, the global illumination data structure may be computed locally (e.g., at the device implementing RESTIR, etc.). In yet another embodiment, the global illumination data structure may be computed remotely (e.g., at a cloud-based computing system separate from the device implementing RESTIR, etc.), and may be sent to a local device implementing RESTIR (e.g., a mobile device, etc.) in order to accelerate the rendering computations performed at the local device.

Also, in one embodiment, when creating an RTXGI data structure, RESTIR may be used to calculate irradiance probes with visibility information. For example, the visibility information may be determined during runtime. In another embodiment, RESTIR may also be used across all surfaces of light probes during the creation of the global illumination data structure (e.g., the RTXGI data structure).

In yet another embodiment, reservoir-based spatiotemporal importance resampling may be performed utilizing a parallel processing unit (PPU) such as the PPU **200** illustrated in FIG. **2**.

More illustrative information will now be set forth regarding various optional architectures and features with which the foregoing framework may be implemented, per the desires of the user. It should be strongly noted that the following information is set forth for illustrative purposes and should not be construed as limiting in any manner. Any of the following features may be optionally incorporated with or without the exclusion of other features described.

Parallel Processing Architecture

FIG. **2** illustrates a parallel processing unit (PPU) **200**, in accordance with an embodiment. In an embodiment, the PPU **200** is a multi-threaded processor that is implemented on one or more integrated circuit devices. The PPU **200** is a latency hiding architecture designed to process many threads in parallel. A thread (i.e., a thread of execution) is an instantiation of a set of instructions configured to be executed by the PPU **200**. In an embodiment, the PPU **200** is a graphics processing unit (GPU) configured to implement a graphics rendering pipeline for processing three-dimensional (3D) graphics data in order to generate two-dimensional (2D) image data for display on a display device such as a liquid crystal display (LCD) device. In other embodiments, the PPU **200** may be utilized for performing general-purpose computations. While one exemplary parallel processor is provided herein for illustrative purposes, it should be strongly noted that such processor is set forth for illustrative purposes only, and that any processor may be employed to supplement and/or substitute for the same.

One or more PPUs **200** may be configured to accelerate thousands of High Performance Computing (HPC), data center, and machine learning applications. The PPU **200** may be configured to accelerate numerous deep learning systems and applications including autonomous vehicle platforms, deep learning, high-accuracy speech, image, and text recognition systems, intelligent video analytics, molecular simulations, drug discovery, disease diagnosis, weather forecasting, big data analytics, astronomy, molecular dynamics simulation, financial modeling, robotics, factory automation, real-time language translation, online search optimizations, and personalized user recommendations, and the like.

As shown in FIG. **2**, the PPU **200** includes an Input/Output (I/O) unit **205**, a front end unit **215**, a scheduler unit **220**, a work distribution unit **225**, a hub **230**, a crossbar (Xbar) **270**, one or more general processing clusters (GPCs) **250**, and one or more partition units **280**. The PPU **200** may be connected to a host processor or other PPUs **200** via one or more high-speed NVLink **210** interconnect. The PPU **200** may be connected to a host processor or other peripheral devices via an interconnect **202**. The PPU **200** may also be connected to a local memory comprising a number of memory devices **204**. In an embodiment, the local memory may comprise a number of dynamic random access memory (DRAM) devices. The DRAM devices may be configured as a high-bandwidth memory (HBM) subsystem, with multiple DRAM dies stacked within each device.

The NVLink **210** interconnect enables systems to scale and include one or more PPUs **200** combined with one or more CPUs, supports cache coherence between the PPUs **200** and CPUs, and CPU mastering. Data and/or commands may be transmitted by the NVLink **210** through the hub **230**

to/from other units of the PPU **200** such as one or more copy engines, a video encoder, a video decoder, a power management unit, etc. (not explicitly shown). The NVLink **210** is described in more detail in conjunction with FIG. **4**B.

The I/O unit **205** is configured to transmit and receive communications (i.e., commands, data, etc.) from a host processor (not shown) over the interconnect **202**. The I/O unit **205** may communicate with the host processor directly via the interconnect **202** or through one or more intermediate devices such as a memory bridge. In an embodiment, the I/O unit **205** may communicate with one or more other processors, such as one or more the PPUs **200** via the interconnect **202**. In an embodiment, the I/O unit **205** implements a Peripheral Component Interconnect Express (PCIe) interface for communications over a PCIe bus and the interconnect **202** is a PCIe bus. In alternative embodiments, the I/O unit **205** may implement other types of well-known interfaces for communicating with external devices.

The I/O unit **205** decodes packets received via the interconnect **202**. In an embodiment, the packets represent commands configured to cause the PPU **200** to perform various operations. The I/O unit **205** transmits the decoded commands to various other units of the PPU **200** as the commands may specify. For example, some commands may be transmitted to the front end unit **215**. Other commands may be transmitted to the hub **230** or other units of the PPU **200** such as one or more copy engines, a video encoder, a video decoder, a power management unit, etc. (not explicitly shown). In other words, the I/O unit **205** is configured to route communications between and among the various logical units of the PPU **200**.

In an embodiment, a program executed by the host processor encodes a command stream in a buffer that provides workloads to the PPU **200** for processing. A workload may comprise several instructions and data to be processed by those instructions. The buffer is a region in a memory that is accessible (i.e., read/write) by both the host processor and the PPU **200**. For example, the I/O unit **205** may be configured to access the buffer in a system memory connected to the interconnect **202** via memory requests transmitted over the interconnect **202**. In an embodiment, the host processor writes the command stream to the buffer and then transmits a pointer to the start of the command stream to the PPU **200**. The front end unit **215** receives pointers to one or more command streams. The front end unit **215** manages the one or more streams, reading commands from the streams and forwarding commands to the various units of the PPU **200**.

The front end unit **215** is coupled to a scheduler unit **220** that configures the various GPCs **250** to process tasks defined by the one or more streams. The scheduler unit **220** is configured to track state information related to the various tasks managed by the scheduler unit **220**. The state may indicate which GPC **250** a task is assigned to, whether the task is active or inactive, a priority level associated with the task, and so forth. The scheduler unit **220** manages the execution of a plurality of tasks on the one or more GPCs **250**.

The scheduler unit **220** is coupled to a work distribution unit **225** that is configured to dispatch tasks for execution on the GPCs **250**. The work distribution unit **225** may track a number of scheduled tasks received from the scheduler unit **220**. In an embodiment, the work distribution unit **225** manages a pending task pool and an active task pool for each of the GPCs **250**. The pending task pool may comprise a number of slots (e.g., 32 slots) that contain tasks assigned to be processed by a particular GPC **250**. The active task pool may comprise a number of slots (e.g., 4 slots) for tasks that are actively being processed by the GPCs **250**. As a GPC **250** finishes the execution of a task, that task is evicted from the active task pool for the GPC **250** and one of the other tasks from the pending task pool is selected and scheduled for execution on the GPC **250**. If an active task has been idle on the GPC **250**, such as while waiting for a data dependency to be resolved, then the active task may be evicted from the GPC **250** and returned to the pending task pool while another task in the pending task pool is selected and scheduled for execution on the GPC **250**.

The work distribution unit **225** communicates with the one or more GPCs **250** via XBar **270**. The XBar **270** is an interconnect network that couples many of the units of the PPU **200** to other units of the PPU **200**. For example, the XBar **270** may be configured to couple the work distribution unit **225** to a particular GPC **250**. Although not shown explicitly, one or more other units of the PPU **200** may also be connected to the XBar **270** via the hub **230**.

The tasks are managed by the scheduler unit **220** and dispatched to a GPC **250** by the work distribution unit **225**. The GPC **250** is configured to process the task and generate results. The results may be consumed by other tasks within the GPC **250**, routed to a different GPC **250** via the XBar **270**, or stored in the memory **204**. The results can be written to the memory **204** via the partition units **280**, which implement a memory interface for reading and writing data to/from the memory **204**. The results can be transmitted to another PPU **200** or CPU via the NVLink **210**. In an embodiment, the PPU **200** includes a number U of partition units **280** that is equal to the number of separate and distinct memory devices **204** coupled to the PPU **200**. A partition unit **280** will be described in more detail below in conjunction with FIG. **3**B.

In an embodiment, a host processor executes a driver kernel that implements an application programming interface (API) that enables one or more applications executing on the host processor to schedule operations for execution on the PPU **200**. In an embodiment, multiple compute applications are simultaneously executed by the PPU **200** and the PPU **200** provides isolation, quality of service (QoS), and independent address spaces for the multiple compute applications. An application may generate instructions (i.e., API calls) that cause the driver kernel to generate one or more tasks for execution by the PPU **200**. The driver kernel outputs tasks to one or more streams being processed by the PPU **200**. Each task may comprise one or more groups of related threads, referred to herein as a warp. In an embodiment, a warp comprises 32 related threads that may be executed in parallel. Cooperating threads may refer to a plurality of threads including instructions to perform the task and that may exchange data through shared memory. Threads and cooperating threads are described in more detail in conjunction with FIG. **4**A.

FIG. **3**A illustrates a GPC **250** of the PPU **200** of FIG. **2**, in accordance with an embodiment. As shown in FIG. **3**A, each GPC **250** includes a number of hardware units for processing tasks. In an embodiment, each GPC **250** includes a pipeline manager **310**, a pre-raster operations unit (PROP) **315**, a raster engine **325**, a work distribution crossbar (WDX) **380**, a memory management unit (MMU) **390**, and one or more Data Processing Clusters (DPCs) **320**. It will be appreciated that the GPC **250** of FIG. **3**A may include other hardware units in lieu of or in addition to the units shown in FIG. **3**A.

In an embodiment, the operation of the GPC **250** is controlled by the pipeline manager **310**. The pipeline man-

ager **310** manages the configuration of the one or more DPCs **320** for processing tasks allocated to the GPC **250**. In an embodiment, the pipeline manager **310** may configure at least one of the one or more DPCs **320** to implement at least a portion of a graphics rendering pipeline. For example, a DPC **320** may be configured to execute a vertex shader program on the programmable streaming multiprocessor (SM) **340**. The pipeline manager **310** may also be configured to route packets received from the work distribution unit **225** to the appropriate logical units within the GPC **250**. For example, some packets may be routed to fixed function hardware units in the PROP **315** and/or raster engine **325** while other packets may be routed to the DPCs **320** for processing by the primitive engine **335** or the SM **340**. In an embodiment, the pipeline manager **310** may configure at least one of the one or more DPCs **320** to implement a neural network model and/or a computing pipeline.

The PROP unit **315** is configured to route data generated by the raster engine **325** and the DPCs **320** to a Raster Operations (ROP) unit, described in more detail in conjunction with FIG. **3**B. The PROP unit **315** may also be configured to perform optimizations for color blending, organize pixel data, perform address translations, and the like.

The raster engine **325** includes a number of fixed function hardware units configured to perform various raster operations. In an embodiment, the raster engine **325** includes a setup engine, a coarse raster engine, a culling engine, a clipping engine, a fine raster engine, and a tile coalescing engine. The setup engine receives transformed vertices and generates plane equations associated with the geometric primitive defined by the vertices. The plane equations are transmitted to the coarse raster engine to generate coverage information (e.g., an x,y coverage mask for a tile) for the primitive. The output of the coarse raster engine is transmitted to the culling engine where fragments associated with the primitive that fail a z-test are culled, and transmitted to a clipping engine where fragments lying outside a viewing frustum are clipped. Those fragments that survive clipping and culling may be passed to the fine raster engine to generate attributes for the pixel fragments based on the plane equations generated by the setup engine. The output of the raster engine **325** comprises fragments to be processed, for example, by a fragment shader implemented within a DPC **320**.

Each DPC **320** included in the GPC **250** includes an M-Pipe Controller (MPC) **330**, a primitive engine **335**, and one or more SMs **340**. The MPC **330** controls the operation of the DPC **320**, routing packets received from the pipeline manager **310** to the appropriate units in the DPC **320**. For example, packets associated with a vertex may be routed to the primitive engine **335**, which is configured to fetch vertex attributes associated with the vertex from the memory **204**. In contrast, packets associated with a shader program may be transmitted to the SM **340**.

The SM **340** comprises a programmable streaming processor that is configured to process tasks represented by a number of threads. Each SM **340** is multi-threaded and configured to execute a plurality of threads (e.g., 32 threads) from a particular group of threads concurrently. In an embodiment, the SM **340** implements a SIMD (Single-Instruction, Multiple-Data) architecture where each thread in a group of threads (i.e., a warp) is configured to process a different set of data based on the same set of instructions. All threads in the group of threads execute the same instructions. In another embodiment, the SM **340** implements a SIMT (Single-Instruction, Multiple Thread) architecture where each thread in a group of threads is configured to process a different set of data based on the same set of instructions, but where individual threads in the group of threads are allowed to diverge during execution. In an embodiment, a program counter, call stack, and execution state is maintained for each warp, enabling concurrency between warps and serial execution within warps when threads within the warp diverge. In another embodiment, a program counter, call stack, and execution state is maintained for each individual thread, enabling equal concurrency between all threads, within and between warps. When execution state is maintained for each individual thread, threads executing the same instructions may be converged and executed in parallel for maximum efficiency. The SM **340** will be described in more detail below in conjunction with FIG. **4**A.

The MMU **390** provides an interface between the GPC **250** and the partition unit **280**. The MMU **390** may provide translation of virtual addresses into physical addresses, memory protection, and arbitration of memory requests. In an embodiment, the MMU **390** provides one or more translation lookaside buffers (TLB s) for performing translation of virtual addresses into physical addresses in the memory **204**.

FIG. **3**B illustrates a memory partition unit **280** of the PPU **200** of FIG. **2**, in accordance with an embodiment. As shown in FIG. **3**B, the memory partition unit **280** includes a Raster Operations (ROP) unit **350**, a level two (L2) cache **360**, and a memory interface **370**. The memory interface **370** is coupled to the memory **204**. Memory interface **370** may implement 32, 64, 128, 1024-bit data buses, or the like, for high-speed data transfer. In an embodiment, the PPU **200** incorporates U memory interfaces **370**, one memory interface **370** per pair of partition units **280**, where each pair of partition units **280** is connected to a corresponding memory device **204**. For example, PPU **200** may be connected to up to Y memory devices **204**, such as high bandwidth memory stacks or graphics double-data-rate, version **5**, synchronous dynamic random access memory, or other types of persistent storage.

In an embodiment, the memory interface **370** implements an HBM2 memory interface and Y equals half U. In an embodiment, the HBM2 memory stacks are located on the same physical package as the PPU **200**, providing substantial power and area savings compared with conventional GDDR5 SDRAM systems. In an embodiment, each HBM2 stack includes four memory dies and Y equals 4, with HBM2 stack including two 128-bit channels per die for a total of 8 channels and a data bus width of 1024 bits.

In an embodiment, the memory **204** supports Single-Error Correcting Double-Error Detecting (SECDED) Error Correction Code (ECC) to protect data. ECC provides higher reliability for compute applications that are sensitive to data corruption. Reliability is especially important in large-scale cluster computing environments where PPUs **200** process very large datasets and/or run applications for extended periods.

In an embodiment, the PPU **200** implements a multi-level memory hierarchy. In an embodiment, the memory partition unit **280** supports a unified memory to provide a single unified virtual address space for CPU and PPU **200** memory, enabling data sharing between virtual memory systems. In an embodiment the frequency of accesses by a PPU **200** to memory located on other processors is traced to ensure that memory pages are moved to the physical memory of the PPU **200** that is accessing the pages more frequently. In an embodiment, the NVLink **210** supports address translation

services allowing the PPU **200** to directly access a CPU's page tables and providing full access to CPU memory by the PPU **200**.

In an embodiment, copy engines transfer data between multiple PPUs **200** or between PPUs **200** and CPUs. The copy engines can generate page faults for addresses that are not mapped into the page tables. The memory partition unit **280** can then service the page faults, mapping the addresses into the page table, after which the copy engine can perform the transfer. In a conventional system, memory is pinned (i.e., non-pageable) for multiple copy engine operations between multiple processors, substantially reducing the available memory. With hardware page faulting, addresses can be passed to the copy engines without worrying if the memory pages are resident, and the copy process is transparent.

Data from the memory **204** or other system memory may be fetched by the memory partition unit **280** and stored in the L2 cache **360**, which is located on-chip and is shared between the various GPCs **250**. As shown, each memory partition unit **280** includes a portion of the L2 cache **360** associated with a corresponding memory device **204**. Lower level caches may then be implemented in various units within the GPCs **250**. For example, each of the SMs **340** may implement a level one (L1) cache. The L1 cache is private memory that is dedicated to a particular SM **340**. Data from the L2 cache **360** may be fetched and stored in each of the L1 caches for processing in the functional units of the SMs **340**. The L2 cache **360** is coupled to the memory interface **370** and the XBar **270**.

The ROP unit **350** performs graphics raster operations related to pixel color, such as color compression, pixel blending, and the like. The ROP unit **350** also implements depth testing in conjunction with the raster engine **325**, receiving a depth for a sample location associated with a pixel fragment from the culling engine of the raster engine **325**. The depth is tested against a corresponding depth in a depth buffer for a sample location associated with the fragment. If the fragment passes the depth test for the sample location, then the ROP unit **350** updates the depth buffer and transmits a result of the depth test to the raster engine **325**. It will be appreciated that the number of partition units **280** may be different than the number of GPCs **250** and, therefore, each ROP unit **350** may be coupled to each of the GPCs **250**. The ROP unit **350** tracks packets received from the different GPCs **250** and determines which GPC **250** that a result generated by the ROP unit **350** is routed to through the Xbar **270**. Although the ROP unit **350** is included within the memory partition unit **280** in FIG. **3**B, in other embodiment, the ROP unit **350** may be outside of the memory partition unit **280**. For example, the ROP unit **350** may reside in the GPC **250** or another unit.

FIG. **4**A illustrates the streaming multi-processor **340** of FIG. **3**A, in accordance with an embodiment. As shown in FIG. **4**A, the SM **340** includes an instruction cache **405**, one or more scheduler units **410**(K), a register file **420**, one or more processing cores **450**, one or more special function units (SFUs) **452**, one or more load/store units (LSUs) **454**, an interconnect network **480**, a shared memory/L1 cache **470**.

As described above, the work distribution unit **225** dispatches tasks for execution on the GPCs **250** of the PPU **200**. The tasks are allocated to a particular DPC **320** within a GPC **250** and, if the task is associated with a shader program, the task may be allocated to an SM **340**. The scheduler unit **410**(K) receives the tasks from the work distribution unit **225** and manages instruction scheduling for one or more thread blocks assigned to the SM **340**. The scheduler unit **410**(K) schedules thread blocks for execution as warps of parallel threads, where each thread block is allocated at least one warp. In an embodiment, each warp executes 32 threads. The scheduler unit **410**(K) may manage a plurality of different thread blocks, allocating the warps to the different thread blocks and then dispatching instructions from the plurality of different cooperative groups to the various functional units (i.e., cores **450**, SFUs **452**, and LSUs **454**) during each clock cycle.

Cooperative Groups is a programming model for organizing groups of communicating threads that allows developers to express the granularity at which threads are communicating, enabling the expression of richer, more efficient parallel decompositions. Cooperative launch APIs support synchronization amongst thread blocks for the execution of parallel algorithms. Conventional programming models provide a single, simple construct for synchronizing cooperating threads: a barrier across all threads of a thread block (i.e., the syncthreads( ) function). However, programmers would often like to define groups of threads at smaller than thread block granularities and synchronize within the defined groups to enable greater performance, design flexibility, and software reuse in the form of collective group-wide function interfaces.

Cooperative Groups enables programmers to define groups of threads explicitly at sub-block (i.e., as small as a single thread) and multi-block granularities, and to perform collective operations such as synchronization on the threads in a cooperative group. The programming model supports clean composition across software boundaries, so that libraries and utility functions can synchronize safely within their local context without having to make assumptions about convergence. Cooperative Groups primitives enable new patterns of cooperative parallelism, including producer-consumer parallelism, opportunistic parallelism, and global synchronization across an entire grid of thread blocks.

A dispatch unit **415** is configured to transmit instructions to one or more of the functional units. In the embodiment, the scheduler unit **410**(K) includes two dispatch units **415** that enable two different instructions from the same warp to be dispatched during each clock cycle. In alternative embodiments, each scheduler unit **410**(K) may include a single dispatch unit **415** or additional dispatch units **415**.

Each SM **340** includes a register file **420** that provides a set of registers for the functional units of the SM **340**. In an embodiment, the register file **420** is divided between each of the functional units such that each functional unit is allocated a dedicated portion of the register file **420**. In another embodiment, the register file **420** is divided between the different warps being executed by the SM **340**. The register file **420** provides temporary storage for operands connected to the data paths of the functional units.

Each SM **340** comprises L processing cores **450**. In an embodiment, the SM **340** includes a large number (e.g., 128, etc.) of distinct processing cores **450**. Each core **450** may include a fully-pipelined, single-precision, double-precision, and/or mixed precision processing unit that includes a floating point arithmetic logic unit and an integer arithmetic logic unit. In an embodiment, the floating point arithmetic logic units implement the IEEE 754-2008 standard for floating point arithmetic. In an embodiment, the cores **450** include 64 single-precision (32-bit) floating point cores, 64 integer cores, 32 double-precision (64-bit) floating point cores, and 8 tensor cores.

Tensor cores configured to perform matrix operations, and, in an embodiment, one or more tensor cores are

included in the cores **450**. In particular, the tensor cores are configured to perform deep learning matrix arithmetic, such as convolution operations for neural network training and inferencing. In an embodiment, each tensor core operates on a 4×4 matrix and performs a matrix multiply and accumulate operation D=A×B+C, where A, B, C, and D are 4×4 matrices.

In an embodiment, the matrix multiply inputs A and B are 16-bit floating point matrices, while the accumulation matrices C and D may be 16-bit floating point or 32-bit floating point matrices. Tensor Cores operate on 16-bit floating point input data with 32-bit floating point accumulation. The 16-bit floating point multiply requires 64 operations and results in a full precision product that is then accumulated using 32-bit floating point addition with the other intermediate products for a 4×4×4 matrix multiply. In practice, Tensor Cores are used to perform much larger two-dimensional or higher dimensional matrix operations, built up from these smaller elements. An API, such as CUDA 9 C++ API, exposes specialized matrix load, matrix multiply and accumulate, and matrix store operations to efficiently use Tensor Cores from a CUDA-C++ program. At the CUDA level, the warp-level interface assumes 16×16 size matrices spanning all 32 threads of the warp.

Each SM **340** also comprises M SFUs **452** that perform special functions (e.g., attribute evaluation, reciprocal square root, and the like). In an embodiment, the SFUs **452** may include a tree traversal unit configured to traverse a hierarchical tree data structure. In an embodiment, the SFUs **452** may include texture unit configured to perform texture map filtering operations. In an embodiment, the texture units are configured to load texture maps (e.g., a 2D array of texels) from the memory **204** and sample the texture maps to produce sampled texture values for use in shader programs executed by the SM **340**. In an embodiment, the texture maps are stored in the shared memory/L1 cache **370**. The texture units implement texture operations such as filtering operations using mip-maps (i.e., texture maps of varying levels of detail). In an embodiment, each SM **240** includes two texture units.

Each SM **340** also comprises N LSUs **454** that implement load and store operations between the shared memory/L1 cache **470** and the register file **420**. Each SM **340** includes an interconnect network **480** that connects each of the functional units to the register file **420** and the LSU **454** to the register file **420**, shared memory/L1 cache **470**. In an embodiment, the interconnect network **480** is a crossbar that can be configured to connect any of the functional units to any of the registers in the register file **420** and connect the LSUs **454** to the register file and memory locations in shared memory/L1 cache **470**.

The shared memory/L1 cache **470** is an array of on-chip memory that allows for data storage and communication between the SM **340** and the primitive engine **335** and between threads in the SM **340**. In an embodiment, the shared memory/L1 cache **470** comprises 128 KB of storage capacity and is in the path from the SM **340** to the partition unit **280**. The shared memory/L1 cache **470** can be used to cache reads and writes. One or more of the shared memory/L1 cache **470**, L2 cache **360**, and memory **204** are backing stores.

Combining data cache and shared memory functionality into a single memory block provides the best overall performance for both types of memory accesses. The capacity is usable as a cache by programs that do not use shared memory. For example, if shared memory is configured to use half of the capacity, texture and load/store operations can use the remaining capacity. Integration within the shared memory/L1 cache **470** enables the shared memory/L1 cache **470** to function as a high-throughput conduit for streaming data while simultaneously providing high-bandwidth and low-latency access to frequently reused data.

When configured for general purpose parallel computation, a simpler configuration can be used compared with graphics processing. Specifically, the fixed function graphics processing units shown in FIG. **2**, are bypassed, creating a much simpler programming model. In the general purpose parallel computation configuration, the work distribution unit **225** assigns and distributes blocks of threads directly to the DPCs **320**. The threads in a block execute the same program, using a unique thread ID in the calculation to ensure each thread generates unique results, using the SM **340** to execute the program and perform calculations, shared memory/L1 cache **470** to communicate between threads, and the LSU **454** to read and write global memory through the shared memory/L1 cache **470** and the memory partition unit **280**. When configured for general purpose parallel computation, the SM **340** can also write commands that the scheduler unit **220** can use to launch new work on the DPCs **320**.

The PPU **200** may be included in a desktop computer, a laptop computer, a tablet computer, servers, supercomputers, a smart-phone (e.g., a wireless, hand-held device), personal digital assistant (PDA), a digital camera, a vehicle, a head mounted display, a hand-held electronic device, and the like. In an embodiment, the PPU **200** is embodied on a single semiconductor substrate. In another embodiment, the PPU **200** is included in a system-on-a-chip (SoC) along with one or more other devices such as additional PPUs **200**, the memory **204**, a reduced instruction set computer (RISC) CPU, a memory management unit (MMU), a digital-to-analog converter (DAC), and the like.

In an embodiment, the PPU **200** may be included on a graphics card that includes one or more memory devices **204**. The graphics card may be configured to interface with a PCIe slot on a motherboard of a desktop computer. In yet another embodiment, the PPU **200** may be an integrated graphics processing unit (iGPU) or parallel processor included in the chipset of the motherboard.

Exemplary Computing System

Systems with multiple GPUs and CPUs are used in a variety of industries as developers expose and leverage more parallelism in applications such as artificial intelligence computing. High-performance GPU-accelerated systems with tens to many thousands of compute nodes are deployed in data centers, research facilities, and supercomputers to solve ever larger problems. As the number of processing devices within the high-performance systems increases, the communication and data transfer mechanisms need to scale to support the increased bandwidth.

FIG. **4**B is a conceptual diagram of a processing system **400** implemented using the PPU **200** of FIG. **2**, in accordance with an embodiment. The exemplary system **465** may be configured to implement the method **100** shown in FIG. **1**. The processing system **400** includes a CPU **430**, switch **410**, and multiple PPUs **200** each and respective memories **204**. The NVLink **210** provides high-speed communication links between each of the PPUs **200**. Although a particular number of NVLink **210** and interconnect **202** connections are illustrated in FIG. **4**B, the number of connections to each PPU **200** and the CPU **430** may vary. The switch **410** interfaces between the interconnect **202** and the CPU **430**.

The PPUs **200**, memories **204**, and NVLinks **210** may be situated on a single semiconductor platform to form a parallel processing module **425**. In an embodiment, the switch **410** supports two or more protocols to interface between various different connections and/or links.

In another embodiment (not shown), the NVLink **210** provides one or more high-speed communication links between each of the PPUs **200** and the CPU **430** and the switch **410** interfaces between the interconnect **202** and each of the PPUs **200**. The PPUs **200**, memories **204**, and interconnect **202** may be situated on a single semiconductor platform to form a parallel processing module **425**. In yet another embodiment (not shown), the interconnect **202** provides one or more communication links between each of the PPUs **200** and the CPU **430** and the switch **410** interfaces between each of the PPUs **200** using the NVLink **210** to provide one or more high-speed communication links between the PPUs **200**. In another embodiment (not shown), the NVLink **210** provides one or more high-speed communication links between the PPUs **200** and the CPU **430** through the switch **410**. In yet another embodiment (not shown), the interconnect **202** provides one or more communication links between each of the PPUs **200** directly. One or more of the NVLink **210** high-speed communication links may be implemented as a physical NVLink interconnect or either an on-chip or on-die interconnect using the same protocol as the NVLink **210**.

In the context of the present description, a single semiconductor platform may refer to a sole unitary semiconductor-based integrated circuit fabricated on a die or chip. It should be noted that the term single semiconductor platform may also refer to multi-chip modules with increased connectivity which simulate on-chip operation and make substantial improvements over utilizing a conventional bus implementation. Of course, the various circuits or devices may also be situated separately or in various combinations of semiconductor platforms per the desires of the user. Alternately, the parallel processing module **425** may be implemented as a circuit board substrate and each of the PPUs **200** and/or memories **204** may be packaged devices. In an embodiment, the CPU **430**, switch **410**, and the parallel processing module **425** are situated on a single semiconductor platform.

In an embodiment, the signaling rate of each NVLink **210** is 20 to 25 Gigabits/second and each PPU **200** includes six NVLink **210** interfaces (as shown in FIG. **4**B, five NVLink **210** interfaces are included for each PPU **200**). Each NVLink **210** provides a data transfer rate of 25 Gigabytes/second in each direction, with six links providing 300 Gigabytes/second. The NVLinks **210** can be used exclusively for PPU-to-PPU communication as shown in FIG. **4**B, or some combination of PPU-to-PPU and PPU-to-CPU, when the CPU **430** also includes one or more NVLink **210** interfaces.

In an embodiment, the NVLink **210** allows direct load/store/atomic access from the CPU **430** to each PPU's **200** memory **204**. In an embodiment, the NVLink **210** supports coherency operations, allowing data read from the memories **204** to be stored in the cache hierarchy of the CPU **430**, reducing cache access latency for the CPU **430**. In an embodiment, the NVLink **210** includes support for Address Translation Services (ATS), allowing the PPU **200** to directly access page tables within the CPU **430**. One or more of the NVLinks **210** may also be configured to operate in a low-power mode.

FIG. **4**C illustrates an exemplary system **465** in which the various architecture and/or functionality of the various previous embodiments may be implemented. The exemplary system **465** may be configured to implement the method **100** shown in FIG. **1**.

As shown, a system **465** is provided including at least one central processing unit **430** that is connected to a communication bus **475**. The communication bus **475** may be implemented using any suitable protocol, such as PCI (Peripheral Component Interconnect), PCI-Express, AGP (Accelerated Graphics Port), HyperTransport, or any other bus or point-to-point communication protocol(s). The system **465** also includes a main memory **440**. Control logic (software) and data are stored in the main memory **440** which may take the form of random access memory (RAM).

The system **465** also includes input devices **460**, the parallel processing system **425**, and display devices **445**, i.e. a conventional CRT (cathode ray tube), LCD (liquid crystal display), LED (light emitting diode), plasma display or the like. User input may be received from the input devices **460**, e.g., keyboard, mouse, touchpad, microphone, and the like. Each of the foregoing modules and/or devices may even be situated on a single semiconductor platform to form the system **465**. Alternately, the various modules may also be situated separately or in various combinations of semiconductor platforms per the desires of the user.

Further, the system **465** may be coupled to a network (e.g., a telecommunications network, local area network (LAN), wireless network, wide area network (WAN) such as the Internet, peer-to-peer network, cable network, or the like) through a network interface **435** for communication purposes.

The system **465** may also include a secondary storage (not shown). The secondary storage includes, for example, a hard disk drive and/or a removable storage drive, representing a floppy disk drive, a magnetic tape drive, a compact disk drive, digital versatile disk (DVD) drive, recording device, universal serial bus (USB) flash memory. The removable storage drive reads from and/or writes to a removable storage unit in a well-known manner.

Computer programs, or computer control logic algorithms, may be stored in the main memory **440** and/or the secondary storage. Such computer programs, when executed, enable the system **465** to perform various functions. The memory **440**, the storage, and/or any other storage are possible examples of computer-readable media.

The architecture and/or functionality of the various previous figures may be implemented in the context of a general computer system, a circuit board system, a game console system dedicated for entertainment purposes, an application-specific system, and/or any other desired system. For example, the system **465** may take the form of a desktop computer, a laptop computer, a tablet computer, servers, supercomputers, a smart-phone (e.g., a wireless, hand-held device), personal digital assistant (PDA), a digital camera, a vehicle, a head mounted display, a hand-held electronic device, a mobile phone device, a television, workstation, game consoles, embedded system, and/or any other type of logic.

While various embodiments have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of a preferred embodiment should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

Graphics Processing Pipeline

In an embodiment, the PPU **200** comprises a graphics processing unit (GPU). The PPU **200** is configured to

receive commands that specify shader programs for processing graphics data. Graphics data may be defined as a set of primitives such as points, lines, triangles, quads, triangle strips, and the like. Typically, a primitive includes data that specifies a number of vertices for the primitive (e.g., in a model-space coordinate system) as well as attributes associated with each vertex of the primitive. The PPU **200** can be configured to process the graphics primitives to generate a frame buffer (i.e., pixel data for each of the pixels of the display).

An application writes model data for a scene (i.e., a collection of vertices and attributes) to a memory such as a system memory or memory **204**. The model data defines each of the objects that may be visible on a display. The application then makes an API call to the driver kernel that requests the model data to be rendered and displayed. The driver kernel reads the model data and writes commands to the one or more streams to perform operations to process the model data. The commands may reference different shader programs to be implemented on the SMs **340** of the PPU **200** including one or more of a vertex shader, hull shader, domain shader, geometry shader, and a pixel shader. For example, one or more of the SMs **340** may be configured to execute a vertex shader program that processes a number of vertices defined by the model data. In an embodiment, the different SMs **340** may be configured to execute different shader programs concurrently. For example, a first subset of SMs **340** may be configured to execute a vertex shader program while a second subset of SMs **340** may be configured to execute a pixel shader program. The first subset of SMs **340** processes vertex data to produce processed vertex data and writes the processed vertex data to the L2 cache **360** and/or the memory **204**. After the processed vertex data is rasterized (i.e., transformed from three-dimensional data into two-dimensional data in screen space) to produce fragment data, the second subset of SMs **340** executes a pixel shader to produce processed fragment data, which is then blended with other processed fragment data and written to the frame buffer in memory **204**. The vertex shader program and pixel shader program may execute concurrently, processing different data from the same scene in a pipelined fashion until all of the model data for the scene has been rendered to the frame buffer. Then, the contents of the frame buffer are transmitted to a display controller for display on a display device.

FIG. **5** is a conceptual diagram of a graphics processing pipeline **500** implemented by the PPU **200** of FIG. **2**, in accordance with an embodiment. The graphics processing pipeline **500** is an abstract flow diagram of the processing steps implemented to generate 2D computer-generated images from 3D geometry data. As is well-known, pipeline architectures may perform long latency operations more efficiently by splitting up the operation into a plurality of stages, where the output of each stage is coupled to the input of the next successive stage. Thus, the graphics processing pipeline **500** receives input data **501** that is transmitted from one stage to the next stage of the graphics processing pipeline **500** to generate output data **502**. In an embodiment, the graphics processing pipeline **500** may represent a graphics processing pipeline defined by the OpenGL® API. As an option, the graphics processing pipeline **500** may be implemented in the context of the functionality and architecture of the previous Figures and/or any subsequent Figure(s).

As shown in FIG. **5**, the graphics processing pipeline **500** comprises a pipeline architecture that includes a number of stages. The stages include, but are not limited to, a data assembly stage **510**, a vertex shading stage **520**, a primitive assembly stage **530**, a geometry shading stage **540**, a viewport scale, cull, and clip (VSCC) stage **550**, a rasterization stage **560**, a fragment shading stage **570**, and a raster operations stage **580**. In an embodiment, the input data **501** comprises commands that configure the processing units to implement the stages of the graphics processing pipeline **500** and geometric primitives (e.g., points, lines, triangles, quads, triangle strips or fans, etc.) to be processed by the stages. The output data **502** may comprise pixel data (i.e., color data) that is copied into a frame buffer or other type of surface data structure in a memory.

The data assembly stage **510** receives the input data **501** that specifies vertex data for high-order surfaces, primitives, or the like. The data assembly stage **510** collects the vertex data in a temporary storage or queue, such as by receiving a command from the host processor that includes a pointer to a buffer in memory and reading the vertex data from the buffer. The vertex data is then transmitted to the vertex shading stage **520** for processing.

The vertex shading stage **520** processes vertex data by performing a set of operations (i.e., a vertex shader or a program) once for each of the vertices. Vertices may be, e.g., specified as a 4-coordinate vector (i.e., <x, y, z, w>) associated with one or more vertex attributes (e.g., color, texture coordinates, surface normal, etc.). The vertex shading stage **520** may manipulate individual vertex attributes such as position, color, texture coordinates, and the like. In other words, the vertex shading stage **520** performs operations on the vertex coordinates or other vertex attributes associated with a vertex. Such operations commonly including lighting operations (i.e., modifying color attributes for a vertex) and transformation operations (i.e., modifying the coordinate space for a vertex). For example, vertices may be specified using coordinates in an object-coordinate space, which are transformed by multiplying the coordinates by a matrix that translates the coordinates from the object-coordinate space into a world space or a normalized-device-coordinate (NCD) space. The vertex shading stage **520** generates transformed vertex data that is transmitted to the primitive assembly stage **530**.

The primitive assembly stage **530** collects vertices output by the vertex shading stage **520** and groups the vertices into geometric primitives for processing by the geometry shading stage **540**. For example, the primitive assembly stage **530** may be configured to group every three consecutive vertices as a geometric primitive (i.e., a triangle) for transmission to the geometry shading stage **540**. In some embodiments, specific vertices may be reused for consecutive geometric primitives (e.g., two consecutive triangles in a triangle strip may share two vertices). The primitive assembly stage **530** transmits geometric primitives (i.e., a collection of associated vertices) to the geometry shading stage **540**.

The geometry shading stage **540** processes geometric primitives by performing a set of operations (i.e., a geometry shader or program) on the geometric primitives. Tessellation operations may generate one or more geometric primitives from each geometric primitive. In other words, the geometry shading stage **540** may subdivide each geometric primitive into a finer mesh of two or more geometric primitives for processing by the rest of the graphics processing pipeline **500**. The geometry shading stage **540** transmits geometric primitives to the viewport SCC stage **550**.

In an embodiment, the graphics processing pipeline **500** may operate within a streaming multiprocessor and the vertex shading stage **520**, the primitive assembly stage **530**, the geometry shading stage **540**, the fragment shading stage **570**, and/or hardware/software associated therewith, may

sequentially perform processing operations. Once the sequential processing operations are complete, in an embodiment, the viewport SCC stage **550** may utilize the data. In an embodiment, primitive data processed by one or more of the stages in the graphics processing pipeline **500** may be written to a cache (e.g. L1 cache, a vertex cache, etc.). In this case, in an embodiment, the viewport SCC stage **550** may access the data in the cache. In an embodiment, the viewport SCC stage **550** and the rasterization stage **560** are implemented as fixed function circuitry.

The viewport SCC stage **550** performs viewport scaling, culling, and clipping of the geometric primitives. Each surface being rendered to is associated with an abstract camera position. The camera position represents a location of a viewer looking at the scene and defines a viewing frustum that encloses the objects of the scene. The viewing frustum may include a viewing plane, a rear plane, and four clipping planes. Any geometric primitive entirely outside of the viewing frustum may be culled (i.e., discarded) because the geometric primitive will not contribute to the final rendered scene. Any geometric primitive that is partially inside the viewing frustum and partially outside the viewing frustum may be clipped (i.e., transformed into a new geometric primitive that is enclosed within the viewing frustum. Furthermore, geometric primitives may each be scaled based on a depth of the viewing frustum. All potentially visible geometric primitives are then transmitted to the rasterization stage **560**.

The rasterization stage **560** converts the 3D geometric primitives into 2D fragments (e.g. capable of being utilized for display, etc.). The rasterization stage **560** may be configured to utilize the vertices of the geometric primitives to setup a set of plane equations from which various attributes can be interpolated. The rasterization stage **560** may also compute a coverage mask for a plurality of pixels that indicates whether one or more sample locations for the pixel intercept the geometric primitive. In an embodiment, z-testing may also be performed to determine if the geometric primitive is occluded by other geometric primitives that have already been rasterized. The rasterization stage **560** generates fragment data (i.e., interpolated vertex attributes associated with a particular sample location for each covered pixel) that are transmitted to the fragment shading stage **570**.

The fragment shading stage **570** processes fragment data by performing a set of operations (i.e., a fragment shader or a program) on each of the fragments. The fragment shading stage **570** may generate pixel data (i.e., color values) for the fragment such as by performing lighting operations or sampling texture maps using interpolated texture coordinates for the fragment. The fragment shading stage **570** generates pixel data that is transmitted to the raster operations stage **580**.

The raster operations stage **580** may perform various operations on the pixel data such as performing alpha tests, stencil tests, and blending the pixel data with other pixel data corresponding to other fragments associated with the pixel. When the raster operations stage **580** has finished processing the pixel data (i.e., the output data **502**), the pixel data may be written to a render target such as a frame buffer, a color buffer, or the like.

It will be appreciated that one or more additional stages may be included in the graphics processing pipeline **500** in addition to or in lieu of one or more of the stages described above. Various implementations of the abstract graphics processing pipeline may implement different stages. Furthermore, one or more of the stages described above may be excluded from the graphics processing pipeline in some embodiments (such as the geometry shading stage **540**). Other types of graphics processing pipelines are contemplated as being within the scope of the present disclosure. Furthermore, any of the stages of the graphics processing pipeline **500** may be implemented by one or more dedicated hardware units within a graphics processor such as PPU **200**. Other stages of the graphics processing pipeline **500** may be implemented by programmable hardware units such as the SM **340** of the PPU **200**.

The graphics processing pipeline **500** may be implemented via an application executed by a host processor, such as a CPU. In an embodiment, a device driver may implement an application programming interface (API) that defines various functions that can be utilized by an application in order to generate graphical data for display. The device driver is a software program that includes a plurality of instructions that control the operation of the PPU **200**. The API provides an abstraction for a programmer that lets a programmer utilize specialized graphics hardware, such as the PPU **200**, to generate the graphical data without requiring the programmer to utilize the specific instruction set for the PPU **200**. The application may include an API call that is routed to the device driver for the PPU **200**. The device driver interprets the API call and performs various operations to respond to the API call. In some instances, the device driver may perform operations by executing instructions on the CPU. In other instances, the device driver may perform operations, at least in part, by launching operations on the PPU **200** utilizing an input/output interface between the CPU and the PPU **200**. In an embodiment, the device driver is configured to implement the graphics processing pipeline **500** utilizing the hardware of the PPU **200**.

Various programs may be executed within the PPU **200** in order to implement the various stages of the graphics processing pipeline **500**. For example, the device driver may launch a kernel on the PPU **200** to perform the vertex shading stage **520** on one SM **340** (or multiple SMs **340**). The device driver (or the initial kernel executed by the PPU **300**) may also launch other kernels on the PPU **300** to perform other stages of the graphics processing pipeline **500**, such as the geometry shading stage **540** and the fragment shading stage **570**. In addition, some of the stages of the graphics processing pipeline **500** may be implemented on fixed unit hardware such as a rasterizer or a data assembler implemented within the PPU **300**. It will be appreciated that results from one kernel may be processed by one or more intervening fixed function hardware units before being processed by a subsequent kernel on an SM **340**.

Example Game Streaming System

Now referring to FIG. **6**, FIG. **6** is an example system diagram for a game streaming system **600**, in accordance with some embodiments of the present disclosure. FIG. **6** includes game server(s) **602** (which may include similar components, features, and/or functionality to the example computing device **700** of FIG. **7**), client device(s) **604** (which may include similar components, features, and/or functionality to the example computing device **700** of FIG. **7**), and network(s) **606** (which may be similar to the network(s) described herein). In some embodiments of the present disclosure, the system **600** may be implemented.

In the system **600**, for a game session, the client device(s) **604** may only receive input data in response to inputs to the input device(s), transmit the input data to the game server(s) **602**, receive encoded display data from the game server(s)

602, and display the display data on the display 624. As such, the more computationally intense computing and processing is offloaded to the game server(s) 602 (e.g., rendering—in particular ray or path tracing—for graphical output of the game session is executed by the GPU(s) of the game server(s) 602). In other words, the game session is streamed to the client device(s) 604 from the game server(s) 602, thereby reducing the requirements of the client device(s) 604 for graphics processing and rendering.

For example, with respect to an instantiation of a game session, a client device 604 may be displaying a frame of the game session on the display 624 based on receiving the display data from the game server(s) 602. The client device 604 may receive an input to one of the input device(s) and generate input data in response. The client device 604 may transmit the input data to the game server(s) 602 via the communication interface 620 and over the network(s) 606 (e.g., the Internet), and the game server(s) 602 may receive the input data via the communication interface 618. The CPU(s) may receive the input data, process the input data, and transmit data to the GPU(s) that causes the GPU(s) to generate a rendering of the game session. For example, the input data may be representative of a movement of a character of the user in a game, firing a weapon, reloading, passing a ball, turning a vehicle, etc. The rendering component 612 may render the game session (e.g., representative of the result of the input data) and the render capture component 614 may capture the rendering of the game session as display data (e.g., as image data capturing the rendered frame of the game session). The rendering of the game session may include ray or path-traced lighting and/or shadow effects, computed using one or more parallel processing units—such as GPUs, which may further employ the use of one or more dedicated hardware accelerators or processing cores to perform ray or path-tracing techniques—of the game server(s) 602. The encoder 616 may then encode the display data to generate encoded display data and the encoded display data may be transmitted to the client device 604 over the network(s) 606 via the communication interface 618. The client device 604 may receive the encoded display data via the communication interface 620 and the decoder 622 may decode the encoded display data to generate the display data. The client device 604 may then display the display data via the display 624.

Example Computing Device

FIG. 7 is a block diagram of an example computing device(s) 700 suitable for use in implementing some embodiments of the present disclosure. Computing device 700 may include an interconnect system 702 that directly or indirectly couples the following devices: memory 704, one or more central processing units (CPUs) 706, one or more graphics processing units (GPUs) 708, a communication interface 710, input/output (I/O) ports 712, input/output components 714, a power supply 716, one or more presentation components 718 (e.g., display(s)), and one or more logic units 720.

Although the various blocks of FIG. 7 are shown as connected via the interconnect system 702 with lines, this is not intended to be limiting and is for clarity only. For example, in some embodiments, a presentation component 718, such as a display device, may be considered an I/O component 714 (e.g., if the display is a touch screen). As another example, the CPUs 706 and/or GPUs 708 may include memory (e.g., the memory 704 may be representative of a storage device in addition to the memory of the GPUs 708, the CPUs 706, and/or other components). In other words, the computing device of FIG. 7 is merely illustrative. Distinction is not made between such categories as "workstation," "server," "laptop," "desktop," "tablet," "client device," "mobile device," "hand-held device," "game console," "electronic control unit (ECU)," "virtual reality system," and/or other device or system types, as all are contemplated within the scope of the computing device of FIG. 7.

The interconnect system 702 may represent one or more links or buses, such as an address bus, a data bus, a control bus, or a combination thereof. The interconnect system 702 may include one or more bus or link types, such as an industry standard architecture (ISA) bus, an extended industry standard architecture (EISA) bus, a video electronics standards association (VESA) bus, a peripheral component interconnect (PCI) bus, a peripheral component interconnect express (PCIe) bus, and/or another type of bus or link. In some embodiments, there are direct connections between components. As an example, the CPU 706 may be directly connected to the memory 704. Further, the CPU 706 may be directly connected to the GPU 708. Where there is direct, or point-to-point connection between components, the interconnect system 702 may include a PCIe link to carry out the connection. In these examples, a PCI bus need not be included in the computing device 700.

The memory 704 may include any of a variety of computer-readable media. The computer-readable media may be any available media that may be accessed by the computing device 700. The computer-readable media may include both volatile and nonvolatile media, and removable and non-removable media. By way of example, and not limitation, the computer-readable media may comprise computer-storage media and communication media.

The computer-storage media may include both volatile and nonvolatile media and/or removable and non-removable media implemented in any method or technology for storage of information such as computer-readable instructions, data structures, program modules, and/or other data types. For example, the memory 704 may store computer-readable instructions (e.g., that represent a program(s) and/or a program element(s), such as an operating system. Computer-storage media may include, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which may be used to store the desired information and which may be accessed by computing device 700. As used herein, computer storage media does not comprise signals per se.

The computer storage media may embody computer-readable instructions, data structures, program modules, and/or other data types in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term "modulated data signal" may refer to a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, the computer storage media may include wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer-readable media.

The CPU(s) 706 may be configured to execute at least some of the computer-readable instructions to control one or

more components of the computing device **700** to perform one or more of the methods and/or processes described herein. The CPU(s) **706** may each include one or more cores (e.g., one, two, four, eight, twenty-eight, seventy-two, etc.) that are capable of handling a multitude of software threads simultaneously. The CPU(s) **706** may include any type of processor, and may include different types of processors depending on the type of computing device **700** implemented (e.g., processors with fewer cores for mobile devices and processors with more cores for servers). For example, depending on the type of computing device **700**, the processor may be an Advanced RISC Machines (ARM) processor implemented using Reduced Instruction Set Computing (RISC) or an x86 processor implemented using Complex Instruction Set Computing (CISC). The computing device **700** may include one or more CPUs **706** in addition to one or more microprocessors or supplementary co-processors, such as math co-processors.

In addition to or alternatively from the CPU(s) **706**, the GPU(s) **708** may be configured to execute at least some of the computer-readable instructions to control one or more components of the computing device **700** to perform one or more of the methods and/or processes described herein. One or more of the GPU(s) **708** may be an integrated GPU (e.g., with one or more of the CPU(s) **706** and/or one or more of the GPU(s) **708** may be a discrete GPU. In embodiments, one or more of the GPU(s) **708** may be a coprocessor of one or more of the CPU(s) **706**. The GPU(s) **708** may be used by the computing device **700** to render graphics (e.g., 3D graphics) or perform general purpose computations. For example, the GPU(s) **708** may be used for General-Purpose computing on GPUs (GPGPU). The GPU(s) **708** may include hundreds or thousands of cores that are capable of handling hundreds or thousands of software threads simultaneously. The GPU(s) **708** may generate pixel data for output images in response to rendering commands (e.g., rendering commands from the CPU(s) **706** received via a host interface). The GPU(s) **708** may include graphics memory, such as display memory, for storing pixel data or any other suitable data, such as GPGPU data. The display memory may be included as part of the memory **704**. The GPU(s) **708** may include two or more GPUs operating in parallel (e.g., via a link). The link may directly connect the GPUs (e.g., using NVLINK) or may connect the GPUs through a switch (e.g., using NVSwitch). When combined together, each GPU **708** may generate pixel data or GPGPU data for different portions of an output or for different outputs (e.g., a first GPU for a first image and a second GPU for a second image). Each GPU may include its own memory, or may share memory with other GPUs.

In addition to or alternatively from the CPU(s) **706** and/or the GPU(s) **708**, the logic unit(s) **720** may be configured to execute at least some of the computer-readable instructions to control one or more components of the computing device **700** to perform one or more of the methods and/or processes described herein. In embodiments, the CPU(s) **706**, the GPU(s) **708**, and/or the logic unit(s) **720** may discretely or jointly perform any combination of the methods, processes and/or portions thereof. One or more of the logic units **720** may be part of and/or integrated in one or more of the CPU(s) **706** and/or the GPU(s) **708** and/or one or more of the logic units **720** may be discrete components or otherwise external to the CPU(s) **706** and/or the GPU(s) **708**. In embodiments, one or more of the logic units **720** may be a coprocessor of one or more of the CPU(s) **706** and/or one or more of the GPU(s) **708**.

Examples of the logic unit(s) **720** include one or more processing cores and/or components thereof, such as Tensor Cores (TCs), Tensor Processing Units (TPUs), Pixel Visual Cores (PVCs), Vision Processing Units (VPUs), Graphics Processing Clusters (GPCs), Texture Processing Clusters (TPCs), Streaming Multiprocessors (SMs), Tree Traversal Units (TTUs), Artificial Intelligence Accelerators (AIAs), Deep Learning Accelerators (DLAs), Arithmetic-Logic Units (ALUs), Application-Specific Integrated Circuits (ASICs), Floating Point Units (FPUs), input/output (I/O) elements, peripheral component interconnect (PCI) or peripheral component interconnect express (PCIe) elements, and/or the like.

The communication interface **710** may include one or more receivers, transmitters, and/or transceivers that enable the computing device **700** to communicate with other computing devices via an electronic communication network, included wired and/or wireless communications. The communication interface **710** may include components and functionality to enable communication over any of a number of different networks, such as wireless networks (e.g., Wi-Fi, Z-Wave, Bluetooth, Bluetooth LE, ZigBee, etc.), wired networks (e.g., communicating over Ethernet or InfiniBand), low-power wide-area networks (e.g., LoRaWAN, SigFox, etc.), and/or the Internet.

The I/O ports **712** may enable the computing device **700** to be logically coupled to other devices including the I/O components **714**, the presentation component(s) **718**, and/or other components, some of which may be built in to (e.g., integrated in) the computing device **700**. Illustrative I/O components **714** include a microphone, mouse, keyboard, joystick, game pad, game controller, satellite dish, scanner, printer, wireless device, etc. The I/O components **714** may provide a natural user interface (NUI) that processes air gestures, voice, or other physiological inputs generated by a user. In some instances, inputs may be transmitted to an appropriate network element for further processing. An NUI may implement any combination of speech recognition, stylus recognition, facial recognition, biometric recognition, gesture recognition both on screen and adjacent to the screen, air gestures, head and eye tracking, and touch recognition (as described in more detail below) associated with a display of the computing device **700**. The computing device **700** may be include depth cameras, such as stereoscopic camera systems, infrared camera systems, RGB camera systems, touchscreen technology, and combinations of these, for gesture detection and recognition. Additionally, the computing device **700** may include accelerometers or gyroscopes (e.g., as part of an inertia measurement unit (IMU)) that enable detection of motion. In some examples, the output of the accelerometers or gyroscopes may be used by the computing device **700** to render immersive augmented reality or virtual reality.

The power supply **716** may include a hard-wired power supply, a battery power supply, or a combination thereof. The power supply **716** may provide power to the computing device **700** to enable the components of the computing device **700** to operate.

The presentation component(s) **718** may include a display (e.g., a monitor, a touch screen, a television screen, a heads-up-display (HUD), other display types, or a combination thereof), speakers, and/or other presentation components. The presentation component(s) **718** may receive data from other components (e.g., the GPU(s) **708**, the CPU(s) **706**, etc.), and output the data (e.g., as an image, video, sound, etc.).

Example Network Environments

Network environments suitable for use in implementing embodiments of the disclosure may include one or more client devices, servers, network attached storage (NAS), other backend devices, and/or other device types. The client devices, servers, and/or other device types (e.g., each device) may be implemented on one or more instances of the computing device(s) **700** of FIG. **7**—e.g., each device may include similar components, features, and/or functionality of the computing device(s) **700**.

Components of a network environment may communicate with each other via a network(s), which may be wired, wireless, or both. The network may include multiple networks, or a network of networks. By way of example, the network may include one or more Wide Area Networks (WANs), one or more Local Area Networks (LANs), one or more public networks such as the Internet and/or a public switched telephone network (PSTN), and/or one or more private networks. Where the network includes a wireless telecommunications network, components such as a base station, a communications tower, or even access points (as well as other components) may provide wireless connectivity.

Compatible network environments may include one or more peer-to-peer network environments—in which case a server may not be included in a network environment—and one or more client-server network environments—in which case one or more servers may be included in a network environment. In peer-to-peer network environments, functionality described herein with respect to a server(s) may be implemented on any number of client devices.

In at least one embodiment, a network environment may include one or more cloud-based network environments, a distributed computing environment, a combination thereof, etc. A cloud-based network environment may include a framework layer, a job scheduler, a resource manager, and a distributed file system implemented on one or more of servers, which may include one or more core network servers and/or edge servers. A framework layer may include a framework to support software of a software layer and/or one or more application(s) of an application layer. The software or application(s) may respectively include web-based service software or applications. In embodiments, one or more of the client devices may use the web-based service software or applications (e.g., by accessing the service software and/or applications via one or more application programming interfaces (APIs)). The framework layer may be, but is not limited to, a type of free and open-source software web application framework such as that may use a distributed file system for large-scale data processing (e.g., "big data").

A cloud-based network environment may provide cloud computing and/or cloud storage that carries out any combination of computing and/or data storage functions described herein (or one or more portions thereof). Any of these various functions may be distributed over multiple locations from central or core servers (e.g., of one or more data centers that may be distributed across a state, a region, a country, the globe, etc.). If a connection to a user (e.g., a client device) is relatively close to an edge server(s), a core server(s) may designate at least a portion of the functionality to the edge server(s). A cloud-based network environment may be private (e.g., limited to a single organization), may be public (e.g., available to many organizations), and/or a combination thereof (e.g., a hybrid cloud environment).

The client device(s) may include at least some of the components, features, and functionality of the example computing device(s) **700** described herein with respect to FIG. **7**. By way of example and not limitation, a client device may be embodied as a Personal Computer (PC), a laptop computer, a mobile device, a smartphone, a tablet computer, a smart watch, a wearable computer, a Personal Digital Assistant (PDA), an MP3 player, a virtual reality headset, a Global Positioning System (GPS) or device, a video player, a video camera, a surveillance device or system, a vehicle, a boat, a flying vessel, a virtual machine, a drone, a robot, a handheld communications device, a hospital device, a gaming device or system, an entertainment system, a vehicle computer system, an embedded system controller, a remote control, an appliance, a consumer electronic device, a workstation, an edge device, any combination of these delineated devices, or any other suitable device.

Reservoir Importance Resampling for Both Direct and Indirect Lighting Using Cached Indirect

Gathering from pre-computed global illumination can be recast as a direct lighting problem. Current methods, including resampled importance sampling (RIS) and RESTIR, include algorithms especially well-suited to scenes with a large number of lights. In one embodiment, reservoir importance resampling (RIR) recasts global illumination as direct lighting to add indirect lighting to previously direct-only algorithms like RIS and ReSTIR.

Reservoir importance resampling asynchronously computes two elements. The first element is global illumination throughout space (i.e., a "light field") that is stored in a world-space data structure. The second element is global illumination on visible surfaces, which is computed by reservoir importance sampling treating all surfaces as light sources that are themselves emissive or lit by the world-space data structure.

The world-space data structure may be computed using a filtered radiance/irradiance volume algorithm based on light field probes (e.g., RTXGI).

Additionally, in one embodiment, reservoir importance resampling for direct lighting takes a random set of candidate points on light sources, evaluates their potential contribution, and then chooses one of them with a probability of selection proportional to their potential contribution. A visibility (shadow) ray is then sent to that point. For example, a set of candidate points on light sources may be traversed, keeping track of the "best" candidate as determined by Monte Carlo sampling. A ray (or a small, fixed number of rays) may then be sent to the "winning" point.

Further, in one embodiment, a lighting volume may be computed by dividing a three-dimensional (3D) volume of a scene into zones (such as a regular or distorted 3D grid) and storing the light that passes through each vertex (i.e., "light probe") of the zones in every direction. These vertices are generally not on surfaces, and usually some effort is expended to ensure that they are not on surfaces, as it is the light in empty space that is useful for the computation. When shading a 3D point, the incident light may be estimated by interpolating values from nearby probes.

The interpolation process using the dynamic diffuse global illumination/RTXGI method uses additional information about the distance to nearby occluders to avoid interpolating light through walls, and weights the surrounding probes based on their proximity and the orientation of the surface being shaded. For example, at a set of points in a grid precomputed "probes" are interpolated to shade a point on reflected objects.

Further still, in one embodiment, RTXGI may be extended to capture incident radiance filtered with different kernel sizes, stored in a MIP chain. RTXGI may describe the equivalent of irradiance probes with visibility; however, the equivalent of filtered radiance probes with visibility may be used. Visibility information may be incorporated and computed dynamically at runtime.

Also, in one embodiment, when computing the RTXGI data structure itself, RESTIR may be used for shading the ray hit points. Previously, brute force exhaustive direct illumination may be used, along with recursive use of the RTXGI data structure. This may be improved by using RIR across all surfaces (and RTXGI recursively in the process). The re-use and filtering in RIR may be applied across the surfaces of the probes instead of across the surface of the screen.

In addition, in one embodiment, RIR may be extended to treat all surfaces as light sources. When a surface is chosen as a sampling candidate, its intensity is treated as the emitted light (as in RIS) plus the reflected light, which is computed by shading the hit surface using only the RTXGI data structure. There is thus no important distinction between initial light sources and indirect lights in the updated resampling method.

In one embodiment, the RTXGI data structure may compute several quantities that can be used to choose better candidates during RESTIR.

FIG. **8** illustrates an exemplary scene **800** for which illumination gathering is being performed utilizing RESTIR and a global illumination data structure, according to one exemplary embodiment. As shown, the scene **800** includes both light sources **802**A-C as well as instances of geometry **804**A-D.

In one embodiment, the lighting contribution from a random set of points from the light sources **802**A-C may be determined utilizing a predetermined lighting equation within RESTIR. In another embodiment, the lighting contribution from a random set of points from the instances of geometry **804**A-D within the scene **800** may be determined by performing a data lookup within a global illumination data structure prepared for the instances of geometry **804**A-D (instead of utilizing the predetermined lighting equation within RESTIR).

However, in another embodiment, both the lighting contribution from points within the light sources **802**A-C and the lighting contribution from points within the instances of geometry **804**A-D within the scene **800** may be determined utilizing a global illumination data structure prepared for both the light sources **802**A-C and the instances of geometry **804**A-D.

Furthermore, in one embodiment, using RESTIR, one of the random set of points from the light sources **802**A-C or the instances of geometry **804**A-D within the scene **800** may be selected, where a probability of point selection is proportional to the potential lighting contribution of the point. Shading calculations may then be performed on this selected point **804**C by casting a shadow ray **808** from a point being shaded in the scene **806** to the selected point **804**C.

In this way, both direct and indirect lighting may be considered during RESTIR by considering both the light sources **802**A-C and the instances of geometry **804**A-D as sources of illumination during shading.

While various embodiments have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of a preferred embodiment should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

The disclosure may be described in the general context of computer code or machine-usable instructions, including computer-executable instructions such as program modules, being executed by a computer or other machine, such as a personal data assistant or other handheld device. Generally, program modules including routines, programs, objects, components, data structures, etc., refer to code that perform particular tasks or implement particular abstract data types. The disclosure may be practiced in a variety of system configurations, including hand-held devices, consumer electronics, general-purpose computers, more specialty computing devices, etc. The disclosure may also be practiced in distributed computing environments where tasks are performed by remote-processing devices that are linked through a communications network.

As used herein, a recitation of "and/or" with respect to two or more elements should be interpreted to mean only one element, or a combination of elements. For example, "element A, element B, and/or element C" may include only element A, only element B, only element C, element A and element B, element A and element C, element B and element C, or elements A, B, and C. In addition, "at least one of element A or element B" may include at least one of element A, at least one of element B, or at least one of element A and at least one of element B. Further, "at least one of element A and element B" may include at least one of element A, at least one of element B, or at least one of element A and at least one of element B.

The subject matter of the present disclosure is described with specificity herein to meet statutory requirements. However, the description itself is not intended to limit the scope of this disclosure. Rather, the inventors have contemplated that the claimed subject matter might also be embodied in other ways, to include different steps or combinations of steps similar to the ones described in this document, in conjunction with other present or future technologies. Moreover, although the terms "step" and/or "block" may be used herein to connote different elements of methods employed, the terms should not be interpreted as implying any particular order among or between various steps herein disclosed unless and except when the order of individual steps is explicitly described.

What is claimed is:

1. A method comprising:

implementing reservoir importance resampling to compute global illumination on visible surfaces in a scene, utilizing a world-space data structure.

2. The method of claim **1**, further comprising computing the global illumination throughout space for the scene, and storing the computed global illumination in the world-space data structure for the scene.

3. The method of claim **1**, wherein the reservoir importance resampling treats points on reflective objects and lights as candidates during the computation of global illumination.

4. The method of claim **3**, wherein the reservoir importance selects one of the candidates for shading, the shading performed by casting a shadow ray from a point being shaded in the scene to a point corresponding to the selected candidate.

5. The method of claim **1**, wherein the world-space data structure includes an RTX global illumination (RTXGI) data structure.

6. The method of claim 1, wherein an emitted color of reflective candidates is looked up in the world-space data structure.

7. The method of claim 1, wherein a filtered radiance/irradiance volume algorithm is used to compute the world-space data structure.

8. The method of claim 1, wherein the reservoir importance resampling treats the visible surfaces as light sources that are emissive or lit by the world-space data structure.

9. The method of claim 1, wherein the world-space data structure is generated using a RTX Global Illumination (RTXGI) algorithm.

10. The method of claim 9, wherein the RTXGI algorithm computes the global illumination using only one ray per sample.

11. The method of claim 1, wherein the reservoir importance resampling asynchronously computes:

global illumination throughout space for the scene which is stored in the world-space data structure, and

the global illumination on the visible surfaces in the scene by treating all surfaces in the scene as light sources.

12. A system comprising:

a processor that is configured to:

implement reservoir importance resampling to compute global illumination on visible surfaces in a scene, utilizing a world-space data structure.

13. The system of claim 12, where the processor is further configured to compute the global illumination throughout space for the scene, and store the computed global illumination in the world-space data structure for the scene.

14. The system of claim 12, wherein the reservoir importance resampling treats points on reflective objects and lights as candidates during the computation of global illumination.

15. The system of claim 12, wherein the world-space data structure includes an RTX global illumination (RTXGI) data structure.

16. The system of claim 12, wherein an emitted color of reflective candidates is looked up in the world-space data structure.

17. The system of claim 12, wherein a filtered radiance/irradiance volume algorithm is used to compute the world-space data structure.

18. The system of claim 12, wherein the reservoir importance resampling treats the visible surfaces as light sources that are emissive or lit by the world-space data structure.

19. The system of claim 12, wherein the world-space data structure is generated using a RTX Global Illumination (RTXGI) algorithm.

20. The system of claim 19, wherein the RTXGI algorithm computes the global illumination using only one ray per sample.

21. A non-transitory computer-readable storage medium storing instructions that, when executed by a processor, causes the processor to perform steps comprising:

implementing reservoir importance resampling to compute global illumination on visible surfaces in a scene, utilizing a world-space data structure.

22. The non-transitory computer-readable storage medium of claim 21, further comprising computing the global illumination throughout space for the scene, and store the computed global illumination in the world-space data structure for the scene.

* * * * *