(19) Deutsches Patent- und Markenamt

(10) **DE 10 2012 212 639 A1** 2013.02.14

(12) # Offenlegungsschrift

(21) Aktenzeichen: **10 2012 212 639.2**
(22) Anmeldetag: **18.07.2012**
(43) Offenlegungstag: **14.02.2013**

(51) Int Cl.: ***G06F 9/38*** (2012.01)

(30) Unionspriorität:
13/209,189 **12.08.2011** **US**

(71) Anmelder:
Nvidia Corp., Santa Clara, Calif., US

(74) Vertreter:
Dilg Haeusler Schindelmann Patentanwaltsgesellschaft mbH, 80636, München, DE

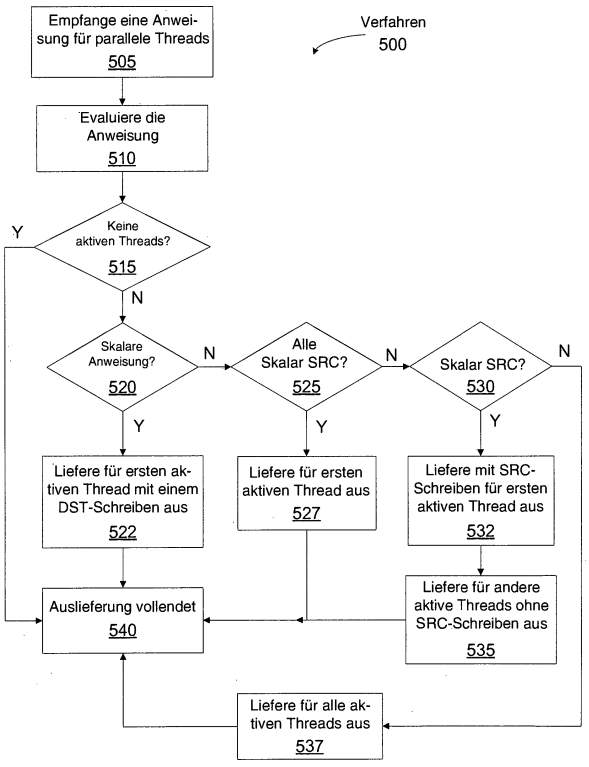
(72) Erfinder:
Krashinsky, Ronny M., San Francisco, Calif., US

Prüfungsantrag gemäß § 44 PatG ist gestellt.

Die folgenden Angaben sind den vom Anmelder eingereichten Unterlagen entnommen

(54) Bezeichnung: **Temporäre SIMT-Ausführungs-Optimierung**

(57) Zusammenfassung: Eine Ausführungsform der vorliegenden Erfindung führt eine Technik zum Optimieren einer Parallel-Thread-Ausführung in einer temporären Einzel-Anweisung-Mehr-Thread-(SIMT)-Architektur aus. Wenn die Threads in einer parallelen Thread-Gruppe temporär auf einer gemeinsamen Verarbeitungs-Pipeline anstatt räumlich auf parallelen Verarbeitungs-Pipeline ausführen, können Ausführungs-Zyklen vermindert werden, wenn einige Threads in der parallelen Thread-Gruppe aufgrund von Divergenz inaktiv sind. Ähnlich kann eine Anweisung zur Ausführung mittels nur eines Threads in der parallelen Thread-Gruppe ausgeliefert werden, wenn die Threads in der parallelen Thread-Gruppe eine skalare Anweisung ausführen. Reduzieren der Anzahl von Threads, welche eine Anweisung ausführen, entfernt unnötige oder redundante Operationen zur Ausführung mittels der Verarbeitungs-Pipelines. Information über skalare Operanden und Operationen und Divergenz der Threads wird in der Anweisungs-Auslieferungs-Logik benutzt, um unnötige oder redundante Aktivität in den Verarbeitungs-Pipelines zu eliminieren.



Beschreibung

HINTERGRUND DER ERFINDUNG

Gebiet der Erfindung

[0001] Die vorliegende Erfindung betrifft im Allgemeinen eine Einzel-Anweisung-Mehr-Thread-(SIMT)-Ausführung und insbesondere Optimierung von temporärer SIMT-Ausführung.

Beschreibung der betreffenden Technik

[0002] Konventionelle SIMT-Mehr-Thread-Prozessoren stellen eine parallele Ausführung von mehreren Threads dadurch bereit, dass Threads in Gruppen organisiert werden und jeder Thread auf einer separaten Verarbeitungs-Pipeline ausgeführt wird. Eine Anweisung zur Ausführung mittels der Threads in einer Gruppe liefert in einem einzelnen Zyklus oder einzelnen Takt (cycle) aus (dispatches). Die Verarbeitungs-Pipeline-Steuersignale sind derart erzeugt, dass alle Threads in einer Gruppe einen ähnlichen Satz von Operation durchführen, wenn die Threads die Stufen (stages) der Verarbeitungs-Pipelines traversieren. Zum Beispiel lesen alle Threads in einer Gruppe Quelloperanden (source operands) von einer Register-Datei, führen die spezifizierte arithmetische Operation in Verarbeitungs-Einheiten durch und schreiben Ergebnisse zurück in die Register-Datei.

[0003] Wenn eine Divergenz zwischen verschiedenen Threads derselben Gruppe erlaubt ist, sind einige der Parallel-Verarbeitungs-Pipelines untätig, während Threads, welche die Verzweigung (branch) nehmen, ausgeführt werden und die übrigen Parallel-Verarbeitungs-Pipelines sind untätig, während die Threads, welche die Verzweigung nicht nahmen, ausgeführt werden. Die Benutzung der parallelen Verarbeitungs-Pipelines kann signifikant vermindert werden, wenn die Ausführung von Threads in einer Gruppe divergiert. Im schlimmsten Falle wird nur ein einzelner Thread zur Ausführung auf den parallelen Verarbeitungs-Pipelines ausgeliefert (dispatched).

[0004] Demgemäß ist, was in der Technik benötigt wird, ein verbessertes System und Verfahren zum Benutzen von Verarbeitungs-Ressourcen in einer Mehr-Thread-Verarbeitungs-Architektur, wenn Threads divergieren können.

ZUSAMMENFASSUNG DER ERFINDUNG

[0005] Eine Ausführungsform der vorliegenden Erfindung führt eine Technik zum Optimieren einer temporären SIMT-Ausführung von parallelen Threads aus. Benutzung von Verarbeitungs-Ressourcen ist verbessert, wenn den Threads erlaubt ist, zu divergieren mittels eines Benutzers einer Mehr-Thread-Verarbeitungs-Architektur mit einer temporären Ausführung verglichen mit einer konventionellen Mehr-Thread-Verarbeitungs-Architektur. Wenn die Threads in einer parallelen Thread-Gruppe temporär auf einer gemeinsamen Verarbeitungs-Pipeline anstatt räumlich auf parallelen Verarbeitungs-Pipelines (eine Verarbeitungs-Pipeline pro Thread) ausführen, können Ausführungs-Zyklen bzw. Ausführungs-Takte vermindert werden, wenn einige Threads in der parallelen Thread-Gruppe aufgrund von Divergenz inaktiv sind. Ähnlich kann eine Anweisung zur Ausführung mittels nur eines Threads in der parallelen Thread-Gruppe ausgeliefert werden, wenn die Threads in der parallelen Thread-Gruppe eine skalare Anweisung ausführen. Information über skalare Operanden und Operationen und Divergenz der Threads wird in der Anweisungs-Auslieferungs-Logik (instruction dispatch logic) benutzt, um unnötige oder redundante Aktivität in den Verarbeitungs-Pipelines zu eliminieren.

[0006] Verschiedene Ausführungsformen eines Verfahrens der Erfindung zum Ausführen einer Anweisung für eine Thread-Gruppe umfassen ein Empfangen der Anweisung zur Ausführung mittels Threads der Thread-Gruppe, Evaluieren der Anweisung, um eine skalare Charakteristik zu identifizieren, und Ausliefern (dispatching) der Anweisung zur Ausführung mittels eines Teils der Threads in der Thread-Gruppe basierend auf der skalaren Charakteristik.

[0007] Ein Vermindern der Anzahl von Threads, welche eine Anweisung ausführen, entfernt unnötige oder redundante Operationen zur Ausführung mittels der Verarbeitungs-Pipelines. Daher ist die Verarbeitungs-Pipeline-Benutzung verbessert und die Performanz kann auch erhöht werden.

KURZE BESCHREIBUNG DER ZEICHNUNGEN

[0008] So dass die Weise, in welcher die oben zitierten Merkmale der vorliegenden Erfindung im Detail verstanden werden können, kann eine besondere Beschreibung der Erfindung, welche kurz oben zusammengefasst ist, durch Bezugnahme auf Ausführungsformen genommen werden, von welchen einige in den angehängten Zeichnungen illustriert sind. Es ist jedoch zu bemerken, dass die angehängten Zeichnungen nur typische Ausführungsformen dieser Erfindung illustrieren und dass sie daher nicht aufzufassen sind, ihren Geltungsbereich zu begrenzen, denn die Erfindung kann andere genauso effektive Ausführungsformen zulassen.

[0009] Fig. 1 ist ein Blockdiagramm, welches ein Computersystem illustriert, welches konfiguriert ist, einen oder mehrere Aspekte der vorliegenden Erfindung zu implementieren;

[0010] Fig. 2 ist ein Blockdiagramm eines Parallel-Verarbeitungs-Subsystem für das Computersystem der **Fig.** 2A, gemäß einer Ausführungsform der vorliegenden Erfindung;

[0011] Fig. 3A ist ein Blockdiagramm eines GPC innerhalb einer PPUs der **Fig.** 2B, gemäß einer Ausführungsform der vorliegenden Erfindung;

[0012] Fig. 3B ist ein Blockdiagramm einer Partitionseinheit innerhalb einer der PPUs der **Fig.** 2B, gemäß einer Ausführungsform der vorliegenden Erfindung;

[0013] Fig. 3C ist ein Blockdiagramm eines Teils des SPM der Fig. 3A, gemäß einer Ausführungsform der vorliegenden Erfindung;

[0014] Fig. 4A ist ein Blockdiagramm eines Teils des SPM von Fig. 3A, welcher für eine temporäre SIMT-Ausführung konfiguriert ist, gemäß einer Ausführungsform der vorliegenden Erfindung;

[0015] Fig. 4B illustriert das Format einer SIMT-Anweisung gemäß einer Ausführungsform der Erfindung;

[0016] Fig. 5A ist ein Flussdiagramm von Verfahrensschritten zum Ausliefern einer Anweisung zur Ausführung mittels einer Thread-Gruppe gemäß einer Ausführungsform der vorliegenden Erfindung;

[0017] Fig. 5B ist ein Flussdiagramm von Verfahrensschritten für einen der Schritte, gezeigt in Fig. 5A, gemäß einer Ausführungsform der vorliegenden Erfindung;

DETAILLIERTE BESCHREIBUNG

[0018] In der folgenden Beschreibung werden zahlreiche spezifische Details ausgeführt, um ein durchgängigeres Verständnis der vorliegenden Erfindung bereitzustellen. Es wird jedoch für den Fachmann in der Technik ersichtlich sein, dass die vorliegende Erfindung ohne ein oder mehrere dieser spezifischen Details praktiziert werden kann. In anderen Fällen sind wohl bekannte Merkmale nicht beschrieben worden, um ein Verschleiern der vorliegenden Erfindung zu vermeiden.

Systemüberblick

[0019] **Fig.** 2A ist ein Blockdiagramm, welches ein Computersystem **100** illustriert, welches konfiguriert ist, einen oder mehrere Aspekte der vorliegenden Erfindung zu implementieren. Computersystem **100** umfasst eine Zentralverarbeitungseinheit (CPU) **102** und einen Systemspeicher **104**, welcher über einen Zwischenverbindungspfad (interconnection path) kommuniziert, welcher eine Speicherbrücke **105** umfassen kann. Speicherbrücke **105**, welche z. B. ein Northbridge-Chip sein kann, ist über einen Bus oder einen anderen Kommunikationspfad **106** (z. B. HyperTransport-Link) mit einer I/O-(Eingabe/Ausgabe)-Brücke **107** verbunden. I/O-Brücke **107**, welche z. B. ein Southbridge-Chip sein kann, empfängt Benutzereingabe von einem oder mehreren Benutzer-Eingabegeräten **108** (z. B. Tastatur, Maus) und leitet die Eingabe an CPU **102** über Pfad **106** und Speicherbrücke **105** weiter. Ein Parallel-Verarbeitungs-Subsystem **112** ist mit der Speicherbrücke **105** über einen Bus oder einen anderen Kommunikationspfad **113** (z. B. einen PCI-Express Accelerated Graphics Port, oder HyperTransport-Link) gekoppelt; in einer Ausführungsform ist das Parallel-Verarbeitungs-Subsystem **112** ein Grafik-Subsystem, welches Pixel an ein Anzeigegerät **110** (z. B. ein konventioneller CRT- oder LCD-basierter Monitor) liefert. Eine Systemplatte **114** ist auch mit der I/O-Brücke **107** verbunden. Ein Switch **116** stellt Verbindungen zwischen I/O-Brücke **107** und anderen Komponenten bereit, wie etwa ein Netzwerkadapter **118** und verschiedenen Hinzufügungskarten (Add-in-Cards) **120** und **121**. Andere Komponenten (nicht explizit ge-

zeigt) umfassend USB- oder andere Port-Verbindungen, CD-Laufwerke, DVD-Laufwerke, Filmaufnahmegeräte, und dergleichen, können auch mit der I/O-Brücke **107** verbunden sein. Kommunikationspfade, welche die verschiedenen Komponenten in **Fig.** 2A wechselseitig verbinden, können unter Benutzung irgendwelcher geeigneten Protokolle implementiert sein, wie etwa PCI (Peripheral Component Interconnect), PCI-Express, AGP (Accelerated Graphics Port), HyperTransport, oder irgendeines oder irgendwelcher Bus- oder Punkt-zu-Punkt-Kommunikations-Protokoll(e), und Verbindungen zwischen verschiedenen Geräten können verschiedene Protokolle benutzen, wie in der Technik bekannt ist.

[0020] In einer Ausführungsform inkorporiert das Parallel-Verarbeitungs-Subsystem **112** Schaltung, welche für Grafik- und Video-Verarbeitung optimiert ist, einschließlich zum Beispiel Videoausgabe-Schaltung, und konstituiert eine Grafik-Verarbeitungseinheit (GPU). In einer anderen Ausführungsform umfasst das Parallel-Verarbeitungs-Subsystem **112** Schaltung, welche für Allgemeinzweck-Verarbeitung optimiert ist, während die darunter liegende Computer-Architektur, welche im größeren Detail hierin beschrieben ist, beibehalten ist. In noch einer anderen Ausführungsform kann das Parallel-Verarbeitungs-Subsystem **102** mit einem oder mit mehreren anderen Systemelementen integriert sein, wie etwa der Speicherbrücke **105**, CPU **102** und I/O-Brücke **107**, um ein System auf dem Chip (system on chip) (SoC) zu bilden.

[0021] Es wird geschätzt werden, dass das hierin gezeigte System illustrativ ist und dass Variationen und Modifikationen möglich sind. Die Verbindungstopologie, einschließlich der Anzahl und der Anordnung von Brücken, der Anzahl von CPUs **102**, und der Anzahl von Parallel-Verarbeitungs-Subsystemen **112** kann wie gewünscht modifiziert werden. Zum Beispiel ist in einigen Ausführungsformen Systemspeicher **104** mit CPU **102** direkt gekoppelt anstatt durch eine Brücke, und andere Geräte kommunizieren mit Systemspeicher **104** über Speicherbrücke **105** und CPU **102**. In anderen alternativen Topologien ist das Parallel-Verarbeitungs-Subsystem **112** mit I/O-Brücke **107** oder direkt mit CPU **102** verbunden anstatt mit der Speicherbrücke **105**. In noch anderen Ausführungsformen können die I/O-Brücke **107** und Speicherbrücke **105** in einen einzelnen Chip integriert sein. Große Ausführungsformen können zwei oder mehr CPUs **102** und zwei oder mehr Parallel-Verarbeitungs-Subsysteme **112** umfassen. Die besonderen Komponenten, welche hierin gezeigt sind, sind optional; z. B. könnte irgendeine Anzahl von Hinzufügungskarten oder peripheren Geräten unterstützt sein. In einigen Ausführungsformen ist der Switch **116** eliminiert und der Netzwerkadapter **116** und Hinzufügungskarten **120**, **121** verbinden direkt mit der I/O-Brücke **107**.

[0022] **Fig.** 2B illustriert ein Parallel-Verarbeitungs-Subsystem **112** gemäß einer Ausführungsform der vorliegenden Erfindung. Wie gezeigt, umfasst das Parallel-Verarbeitungs-Subsystem **112** eine oder mehrere Parallel-Verarbeitungseinheiten (PPUs) **202**, wobei jede von diesen mit einem lokalen Parallel-Verarbeitungs-(PP)-Speicher **204** gekoppelt ist. Im Allgemeinen umfasst ein Parallel-Verarbeitungs-Subsystem eine Anzahl U von PPUs, wobei $U \geq 1$ (hierin sind mehrere Instanzen von ähnlichen Objekten mit Referenznummern bezeichnet, welche das Objekt identifizieren und Nummern in Klammern die Instanz identifizieren, wenn benötigt). PPUs **202** und Parallel-Verarbeitungs-Speicher **204** können unter Benutzung von einem oder mehreren integrierte-Schaltung-Geräten implementiert sein, wie etwa programmierbare Prozessoren, Anwendungs-spezifische integrierte Schaltungen (ASICs), oder Speichergeräte, oder in irgendeiner anderen technisch machbaren Weise.

[0023] Mit Bezug wieder auf **Fig.** 2A sind in einigen Ausführungsformen einige oder alle der PPUs **202** in dem Parallel-Verarbeitungs-Subsystem **112** Grafikprozessoren mit Render-Pipelines, welche konfiguriert sein können, um verschiedene Aufgaben durchzuführen, welche das Erzeugen von Pixeldaten von Grafik-Daten, welche mittels CPU **102** und/oder Systemspeicher **104** über Speicherbrücke **105** und Kommunikationspfad **113** zugeführt sind, ein Interagieren mit lokalem Parallel-Verarbeitungs-Speicher **204** (welcher als ein Grafikspeicher benutzt werden kann einschließlich z. B. eines konventionellen Bildpuffers (frame buffer)), um Pixeldaten zu speichern und zu aktualisieren, ein Liefern von Pixeldaten an das Anzeigegeräte **110**, und dergleichen betreffen. In einigen Ausführungsformen kann das Parallel-Verarbeitungs-Subsystem **112** eine oder mehrere PPUs **202** umfassen, welche als Grafikprozessoren operieren, und eine oder mehrere andere PPUs **202**, welche für Allgemeinzweck-Berechnungen benutzt werden können. Die PPUs können identisch sein oder verschieden sein und jede PPU kann sein eigenes dediziertes Parallel-Verarbeitungs-Speichergerät(e) haben oder braucht nicht dedizierte Parallel-Verarbeitungs-Speichergerät(e) zu haben. Eine oder mehrere PPUs **202** können Daten an das Anzeigegeräte **110** ausgeben oder jede PPU **202** kann Daten an eines oder mehrere Anzeigegeräte **110** ausgeben.

[0024] Im Betrieb ist CPU **102** der Master-Prozessor von Computersystems **100**, welcher Operationen von anderen Systemkomponenten steuert und koordiniert. Insbesondere stellt CPU **102** Befehle aus (issues), welche den Betrieb von PPUs **202** steuern. In einigen Ausführungsformen schreibt CPU **102** einen Strom von Befehlen für jede PPU **202** an einen Schiebepuffer oder Push-Puffer (pushbuffer) (nicht explizit in **Fig.** 2A oder **Fig. 3B**

gezeigt), welcher in dem Systemspeicher **104**, Parallel-Verarbeitungs-Speicher **204** oder irgendeiner anderen Speicherstelle, welche für sowohl CPU **102** als auch PPU **202** zugreifbar ist, gespeichert ist. PPU **202** liest den Befehlsstrom von dem Push-Puffer und führt dann Befehle asynchron relativ zu dem Betrieb von CPU **102** aus.

[0025] Mit Bezug nun zurück auf **Fig.** 2B umfasst jede PPU **202** eine I/O-(Eingabe/Ausgabe)-Einheit **205**, welche mit dem Rest des Computersystems **100** über Kommunikationspfad **113** kommuniziert, welcher zu Speicherbrücke **105** (oder in einer anderen Ausführungsform direkt mit CPU **102**) verbindet. Die Verbindung von PPU **202** an den Rest des Computersystems **100** kann auch variiert werden. In einigen Ausführungsformen ist das Parallel-Verarbeitungs-Subsystem **112** als eine Hinzufügungskarte implementiert, welche in einen Erweiterungsschlitz oder Erweiterungssteckplatz (expansion slot) von Computersystem **100** eingeführt werden kann. In anderen Ausführungsformen kann eine PPU **202** auf einem einzelnen Chip integriert sein mit einer Bus-Brücke, wie etwa Speicherbrücke **105** oder I/O-Brücke **107**. In noch anderen Ausführungsformen können einige oder alle Elemente von PPU **202** auf einem einzelnen Chip mit CPU **102** integriert sein.

[0026] In einer Ausführungsform ist der Kommunikationspfad **113** ein PCI-Express-Link, in welchem dedizierte Spuren oder Bahnen (lanes) an jede PPU **202** alloziert sind, wie es in der Technik bekannt ist. Andere Kommunikationspfade können auch benutzt werden. Eine I/O-Einheit **205** erzeugt Pakete (oder andere Signale) für eine Übermittlung auf Kommunikationspfad **113** und empfängt auch alle einlaufenden oder hereinkommenden (incoming) Pakete (oder andere Signale) von Kommunikationspfad **113**, wobei die einlaufenden Pakete zu den geeigneten Komponenten von PPU **202** gerichtet werden. Zum Beispiel können Befehle, welche Verarbeitungs-Aufgaben betreffen, an eine Host-Schnittstelle **206** gerichtet werden, während Befehle, welche Speicher-Operationen betreffen (z. B. Lesen von oder Schreiben auf Parallel-Verarbeitungsspeicher **204**) an eine Speicher-Kreuzschiene-Einheit (memory crossbar unit) **202** gerichtet werden können. Host-Schnittstelle **206** liest jeden Push-Puffer und gibt die Arbeit, welche mittels des Push-Puffers spezifiziert ist, an ein Frontend **212** aus.

[0027] Jede PPU **202** implementiert vorteilhafter Weise eine Hochparallel Verarbeitungs-Architektur. Wie im Detail gezeigt ist, umfasst PPU **202**(0) ein Verarbeitungscluster-Feld (processing cluster array) **230**, welches eine Anzahl C von Allgemein-Verarbeitungs-Clustern (GPCs) **208** umfasst, wobei $C \geq 1$. Jeder GPC **208** ist in der Lage, eine große Anzahl (z. B. Hunderte oder Tausende) von Threads simultan (concurrently) auszuführen, wobei jeder Thread eine Instanz eines Programms ist. In verschiedenen Anwendungen können verschiedene GPCs **208** zur Verarbeitung von verschiedenen Typen von Programmen oder zum Durchführen von verschiedenen Typen von Berechnungen alloziert werden. Zum Beispiel kann in einer Grafik-Anwendung ein erster Satz von GPCs **208** alloziert werden, um Stück-Tessellations-Operationen (patch tessellation operations) durchzuführen und Primitive-Topologien für Patches oder Stücke zu erzeugen, und ein zweiter Satz von GPCs **208** kann alloziert sein, um Tessellations-Schattierung (tessellation shading) durchzuführen, um Patch-Parameter für die Primitive-Topologien zu evaluieren und Vertex-Positionen und andere Pro-Vertex-Attribute zu bestimmen. Die Allozierung von GPCs **208** kann abhängig von der Arbeitsbelastung, welche für jeden Typ von Programm oder Berechnung auftritt, variiert werden.

[0028] GPCs **208** empfangen Verarbeitungs-Aufgaben, welche auszuführen sind, über eine Arbeitsverteilungs-Einheit **200**, welche Befehle, welche Verarbeitungs-Aufgaben definieren, von der Frontend-Einheit **212** empfängt. Verarbeitungs-Aufgaben umfassen Indizes von Daten, welche zu prozessieren sind, z. B. Oberflächen-(Patch)-Daten, Primitive-Daten, Vertex-Daten und/oder Pixel-Daten, sowie Zustands-Parameter und Befehle, welche definieren, wie die Daten zu prozessieren sind (z. B. welches Programm auszuführen ist). Arbeitsverteilungs-Einheit **200** kann konfiguriert sein, die Indizes, welche den Aufgaben entsprechen, zu holen, oder die Arbeitsverteilungs-Einheit **200** kann die Indizes von dem Frontend **212** empfangen. Frontend **212** stellt sicher, dass die GPCs **208** auf einen gültigen Zustand konfiguriert sind, bevor die Verarbeitung, welche mittels der Push-Puffer spezifiziert wird, initiiert wird.

[0029] Wenn PPU **202** zum Beispiel für Grafik-Verarbeitung benutzt wird, ist die Verarbeitungs-Arbeitsbelastung für jeden Patch in ungefähr gleichgroße Aufgaben aufgeteilt, um eine Verteilung der Tessellations-Verarbeitung an mehrere GPCs **208** zu ermöglichen. Eine Arbeits-Verteilungseinheit **200** kann konfiguriert sein, Aufgaben bei einer Frequenz zu erzeugen, was befähigt, die Aufgaben an mehrere GPCs **208** zur Verarbeitung bereitzustellen. Im Gegensatz dazu ist in konventionellen Systemen Verarbeitung typischerweise mittels einer einzigen Verarbeitungs-Maschine (processing engine) durchgeführt, während andere Verarbeitungs-Maschinen untätig bleiben, wobei sie darauf warten, dass die einzelne Verarbeitungs-Maschine ihre Aufgaben komplettiert, bevor sie ihre Verarbeitungs-Aufgaben beginnen. In einigen Ausführungsformen der vorliegenden Erfindung sind Teile von GPCs **208** konfiguriert, verschiedene Typen von Verarbeitung durchzuführen. Zum Beispiel kann ein erster Teil konfiguriert sein, eine Vertex-Schattierung und Topologie-Erzeugung durchzuführen, ein zweiter Teil kann konfiguriert sein, eine Tessellation und Geometrie-Schattierung durchzuführen und

ein dritter Teil kann konfiguriert sein, Pixel-Schattierung in einem Pixel-Raum durchzuführen, um ein gerendertes Bild zu erzeugen. Zwischen-Daten, welche mittels der GPCs **208** produziert sind, können in Puffern gespeichert sein, um zu erlauben, dass die Zwischen-Daten zwischen GPCs **208** für weitere Verarbeitung übermittelt werden.

[0030] Speicher-Schnittstelle **214** umfasst ein Anzahl D von Partitions-Einheiten **215**, welche jeweils direkt mit einem Teil von Parallel-Verarbeitungs-Speicher **204** gekoppelt sind, wobei D ≥ 1. Wie gezeigt, ist die Anzahl von Partitions-Einheiten **215** im Allgemeinen gleich der Anzahl von DRAM **220**. In anderen Ausführungsformen muss die Anzahl von Partitions-Einheiten **215** nicht gleich der Nummer von Speicher-Geräten sein. Fachleute in der Technik werden schätzen, dass DRAM **220** durch irgendwelche anderen geeigneten Speicher-Geräte ersetzt werden kann und von einem im Allgemeinen konventionellen Design sein kann. Eine detaillierte Beschreibung wird daher ausgelassen. Render-Ziele (render targets), wie etwa Frame-Puffer oder Textur-Maps können über DRAMs **220** gespeichert sein, was den Partitions-Einheiten **215** erlaubt, Teile von jedem Render-Target in paralleler Weise zu schreiben, um effektiv die verfügbare Bandbreite von Parallel-Verarbeitungs-Speicher **204** zu nutzen.

[0031] Irgendeine von GPCs **208** kann Daten verarbeiten, welche auf irgendeinen der DRAMs **220** innerhalb des Parallel-Verarbeitungs-Speichers **204** zu schreiben sind. Kreuzschiene-Einheit **210** ist konfiguriert, um die Ausgabe von jedem GPC **208** an den Eingang irgendeiner Partitions-Einheit **215** oder an irgendeinen GPC **208** für weitere Verarbeitung zu leiten (route). GPCs **208** kommunizieren mit der Speicher-Schnittstelle **214** durch die Kreuzschiene **210**, um von/auf verschiedene externe Speicher-Geräte zu schreiben oder zu lesen. In einer Ausführungsform hat die Kreuzschiene-Einheit **210** eine Verbindung der Speicher-Schnittstelle **214**, um mit I/O-Einheit **205** zu kommunizieren, sowie eine Verbindung mit lokalem Parallel-Verarbeitungs-Speicher **204**, um dadurch den Verarbeitungs-Kernen innerhalb der verschiedenen GPCs **208** zu ermöglichen, mit dem System-Speicher **104** oder einem anderen Speicher zu kommunizieren, welcher nicht lokal zu der PPU **202** ist. In der in **Fig.** 2B gezeigten Ausführungsform ist die Kreuzschiene-Einheit **210** direkt mit I/O-Einheit **205** verbunden. Kreuzschiene-Einheit **210** kann virtuelle Kanäle benutzen, um Verkehrsströme zwischen den GPCs **208** und den Partitions-Einheiten **215** zu separieren.

[0032] Wiederum können GPCs **208** programmiert sein, Verarbeitungs-Aufgaben durchzuführen, welche eine große Verschiedenheit von Anwendungen betreffen, einschließlich aber nicht darauf beschränkt, lineare oder nichtlineare Daten-Transformationen, Filtern von Video- und/oder Audio-Daten, Modellierungs-Operationen (z. B. Anwenden der Gesetze der Physik, um Position, Geschwindigkeit und andere Attribute von Objekten zu bestimmen), Bild-Render-Operationen (z. B. Tessellations-Schattierung, Vertex-Schattierung, Geometrie-Schattierung und/oder Pixel-Schattierungs-Programme), usw. PPUs **202** können Daten von dem System-Speicher **104** und/oder Lokal-Parallel-Verarbeitungs-Speichern **204** in internen (On-Chip)-Speicher transferieren, können die Daten prozessieren, und können Ergebnis-Daten zurück in den System-Speicher **104** und/oder lokalen Parallel-Verarbeitungs-Speicher **204** schreiben, wo auf solche Daten mittels anderer System-Komponenten zugegriffen werden kann, einschließlich CPU **102** oder ein anderes Parallel-Verarbeitungs-Subsystem **112**.

[0033] Eine PPU **202** kann mit irgendeiner Menge/Umfang (amount) von Lokal-Parallel-Verarbeitungs-Speicher **204** bereitgestellt sein, einschließlich keines Lokal-Speichers, und kann Lokal-Speicher und System-Speicher in irgendeiner Kombination benutzen. Zum Beispiel kann eine PPU **202** ein Grafikprozessor in einer unifizierter-Speicher-Architektur (unified memory architecture) (UMA)-Ausführungsform sein. In solchen Ausführungsformen würde wenig oder kein dedizierter Grafik-(Parallel-Verarbeitungs)-Speicher bereitgestellt sein und PPU **202** würde System-Speicher exklusiv oder fast exklusiv benutzen. In UMA-Ausführungsformen kann eine PPU **202** in einen Brücken-Chip oder Prozessor-Chip integriert sein oder als ein diskreter Chip bereitgestellt sein mit einem Hochgeschwindigkeits-Link (z. B. PCI-Express), welcher die PPU **202** mit System-Speicher über einen Brücke-Chip oder ein anderes Kommunikations-Mittel verbindet.

[0034] Wie oben bemerkt ist, kann irgendeine Anzahl von PPUs **202** in einem Parallel-Verarbeitungs-Subsystem **112** umfasst sein. Zum Beispiel können mehrere PPUs **202** auf einer einzelnen Hinzufügungskarte bereitgestellt sein oder mehrere Hinzufügungskarten können mit dem Kommunikationspfad **113** verbunden sein oder eine oder mehrere der PPUs **202** können in einen Brücken-Chip integriert sein. PPUs **202** in einem Mehr-PPU-System können identisch sein oder verschieden voneinander sein. Zum Beispiel könnten verschiedene PPUs **202** verschiedene Anzahlen von Verarbeitungs-Kernen haben, verschiedene Mengen oder Größen von Lokal-Parallel-Verarbeitungs-Speicher, usw. Wo mehrere PPUs **202** vorhanden sind, können diese PPUs in paralleler Weise betrieben werden, um Daten bei einem höheren Durchsatz zu verarbeiten als es mit einer einzelnen PPU **202** möglich. ist. Systeme, welche eine oder mehrere PPUs **202** inkorporieren, können in einer Verschiedenheit von Konfigurationen und Formfaktoren implementiert sein, einschließlich Schreibtisch-Com-

puter, Laptop-Computer, oder handgehaltenen Personal-Computern, Servern, Arbeitsstationen, Spielekonsolen, eingebetteten Systemen und dergleichen.

Verarbeitungs-Cluster-Feld-Überblick

[0035] Fig. 3A ist ein Blockdiagramm eines GPC **208** innerhalb einer der PPUs **202** der **Fig.** 2B, gemäß einer Ausführungsform der vorliegenden Erfindung. Jeder GPC **208** kann konfiguriert sein, eine große Anzahl von Threads parallel auszuführen, wobei der Ausdruck „Thread" sich auf eine Instanz eines bestimmten Programms bezieht, welches auf einem bestimmten Satz von Eingabe-Daten ausführt. In einigen Ausführungsformen werden Einzel-Anweisung-, Mehr-Daten-(SIMD)-Befehls-Ausstellungs-Techniken benutzt, um eine parallele Ausführung einer großen Anzahl von Threads zu unterstützen, ohne mehrere unabhängige Anweisungs-Einheiten bereitzustellen. In anderen Ausführungsformen werden Einzel-Anweisung-, Mehrfach-Thread-(SIMT)-Techniken benutzt, um eine parallele Ausführung einer großen Anzahl von im Allgemeinen synchronisierten Threads zu unterstützen, unter Benutzung einer gemeinsamen Anweisungs-Einheit, welche konfiguriert ist, Anweisungen für einen Satz von Verarbeitungs-Maschinen innerhalb jedes der GPCs **208** auszustellen (issue). Unähnlich zu einem SIMD-Ausführungs-Regime, wobei alle Verarbeitungs-Maschinen typischerweise identische Anweisungen ausführen, erlaubt eine große SIMT-Ausführung verschiedenen Threads, leichter divergenten Ausführungspfaden durch ein gegebenes Thread-Programm zu folgen. Fachleute in der Technik werden verstehen, dass ein SIMD-Verarbeitungs-Regime eine funktionale Untermenge eines SIMT-Verarbeitungs-Regimes repräsentiert.

[0036] Betrieb von GPC **208** wird vorteilhafterweise über einen Pipeline-Manager **305** gesteuert, welcher Verarbeitungs-Aufgaben an Strömungs-Mehrfach-Prozessoren (streaming multiprocessors) (SPMs) **310** verteilt. Pipeline-Manager **305** kann auch konfiguriert sein, eine Arbeitsverteilungs-Kreuzschiene (work distribution crossbar) **330** dadurch zu steuern, dass Ziele (destinations) für prozessierte Daten-Ausgaben mittels SPMs **310** spezifiziert sind.

[0037] In einer Ausführungsform umfasst jede GPC **208** eine Anzahl M von SPMs **310**, wobei $M \geq 1$, wobei jeder SPM **310** konfiguriert ist, eine oder mehrere Thread-Gruppen zu verarbeiten. Auch umfasst jeder SPM **310** vorteilhafterweise einen identischen Satz von funktionalen Ausführungseinheiten (z. B. Ausführungseinheiten und Lade-Speicher-Einheiten – gezeigt als Exec-Einheiten **302** und LSUs **303** in Fig. 3C), welche in einer Pipeline angeordnet sein können (pipelined), was erlaubt, eine neue Anweisung auszustellen, bevor eine vorherige Anweisung beendet worden ist, wie es in der Technik bekannt ist. Irgendeine Kombination von funktionalen Ausführungs-Einheiten kann bereitgestellt sein. In einer Ausführungsform unterstützen die funktionalen Einheiten eine Verschiedenheit von Operationen, einschließlich Ganzzahl-Arithmetik und Gleitzahl-Arithmetik (z. B. Addition und Multiplikation), Vergleichs-Operationen, Bool'sche Operationen (AND, OR, XOR), Bit-Verschiebung und Berechnen von verschiedenen algebraischen Funktionen (z. B. planare Interpolation, trigonometrische, exponentiale und logarithmische Funktionen); und dieselbe Funktional-Einheit-Hardware kann eingesetzt werden, um verschiedene Operationen durchzuführen.

[0038] Die Serie von Anweisungen, welche an eine bestimmte GPC **208** übermittelt wird, konstituiert einen Thread, wie vorher hierin definiert ist, und die Sammlung einer gewissen Anzahl von simultan ausführenden Threads über die Parallel-Verarbeitungs-Maschinen (nicht gezeigt) innerhalb eines SPM **310** wird hierin als ein „Warp" oder „Thread-Gruppe" bezeichnet. Wie hierin benutzt, bezeichnet eine „Thread-Gruppe" eine Gruppe von Threads, welche simultan dasselbe Programm auf verschiedenen Eingabe-Daten ausführen, wobei ein Thread der Gruppe an eine verschiedene Verarbeitungs-Maschine innerhalb eines SPM **310** zugewiesen ist. Eine Thread-Gruppe kann weniger Threads umfassen als die Anzahl von Verarbeitungs-Einheiten innerhalb des SPM **310**, in welchem Fall einige Verarbeitungs-Maschinen während Zyklen untätig sein werden, wenn diese Thread-Gruppe verarbeitet wird. Eine Thread-Gruppe kann auch mehr Threads umfassen als die Anzahl von Verarbeitungs-Maschinen innerhalb des SPM **310**, in welchem Fall die Verarbeitung über nachfolgende Taktzyklen stattfinden wird. Da jeder SPM **310** bis zu G Thread-Gruppen gleichzeitig unterstützen kann, folgt, dass bis zu $G \cdot M$ Thread-Gruppen zu einer gegebenen Zeit in GPC **208** ausführen können.

[0039] Zusätzlich kann eine Mehrzahl von bezogenen Thread-Gruppen aktiv sein, in verschiedenen Phasen einer Ausführung, zu derselben Zeit innerhalb eines SPM **310**. Diese Sammlung von Thread-Gruppen wird hierin als ein „kooperatives Thread-Feld" (cooperative thread array) („CTA") oder „Thread-Feld" bezeichnet. Die Größe eines bestimmten CTA ist $m \cdot k$, wobei k die Anzahl von gleichzeitig ausführenden Threads in einer Thread-Gruppe ist und typischerweise ein ganzzahliges Vielfaches der Anzahl von Parallel-Verarbeitungs-Einheiten innerhalb des SPM **310** ist, und wobei m die Anzahl von Thread-Gruppen ist, welche simultan innerhalb

des SPM **310** aktiv sind. Die Größe eines CTA ist im Allgemeinen mittels des Programmierers bestimmt und mittels der Menge von Hardware-Ressourcen, wie Speicher oder Register, welche für das CTA verfügbar sind.

[0040] Jeder SPM **310** beinhaltet einen L1-Cache (nicht gezeigt) oder benutzt Raum (space) in einem entsprechenden L1-Cache außerhalb des SPM **310**, welcher benutzt ist, um Lade- und Speicher-Operationen durchzuführen. Jeder SPM **310** hat auch Zugriff auf L2-Caches innerhalb der Partitionseinheiten **215**, welche unter allen GPCs **208** gemeinsam benutzt oder geteilt sind (shared) und benutzt werden können, um Daten zwischen Threads zu transferieren. Schließlich haben die SPMs **310** Zugriff auf Off-Chip „globalen" Speicher, welcher z. B. Parallel-Verarbeitungs-Speicher **204** oder System-Speicher **104** umfassen kann. Es ist verstanden, dass irgendein Speicher extern zu PPU **202** als globaler Speicher benutzt werden kann. Zusätzlich kann ein L1.5-Cache **335** innerhalb des GPC **208** umfasst sein, welcher konfiguriert ist, Daten zu empfangen und zu halten, welche von dem Speicher über Speicher-Schnittstelle **214** geholt sind, abgefragt mittels SPM **310**, einschließlich Anweisungen, uniforme Daten und konstante Daten, und die angefragten Daten an SPM **310** bereitzustellen. Ausführungsformen, welche mehrere SPMs **310** in GPC **208** haben, teilen oder benutzen gemeinsam (share) in vorteilhafter Weise gemeinsame Anweisungen und Daten, welche in L1.5-Cache **335** gecached sind.

[0041] Jeder GPC **208** kann eine Speicher-Management-Einheit (MMU) **328** umfassen, welche konfiguriert ist, virtuelle Adressen in physikalische Adressen abzubilden (map). In anderen Ausführungsformen, können MMU(s) **328** innerhalb der Speicher-Schnittstelle **214** ansässig sein (reside). Die MMU **328** umfasst einen Satz von Seite-Tabelle-Einträgen (page table entry) (PTEs), welche benutzt werden, um eine virtuelle Adresse in eine physikalische Adresse einer Kachel (tile) und optional einen Cache-Zeilen-Index abzubilden. Die MMU **328** kann Adresse-Übersetzungs-Puffer (translation lookaside buffer) (TLB) oder Caches umfassen, welche innerhalb des Mehrfach-Prozessors SPM **310** oder dem L1-Cache oder GPC **208** ansässig sein können. Die physikalische Adresse ist verarbeitet, um Oberflächendaten-Zugriffslokalität zu verteilen, um eine effiziente Abfrage-Verschachtelung (interleaving) unter Partitions-Einheiten zu erlauben. Der Cache-Zeile-Index kann benutzt werden, um zu bestimmen, ob eine Anfrage nach einer Cache-Zeile ein Treffer ist oder eine Verfehlung ist oder nicht.

[0042] in Grafik- und Berechnungs-Anwendungen kann eine GPC **208** derart konfiguriert sein, dass jeder SPM **310** mit einer Textur-Einheit **315** zum Durchführen von Textur-Abbildungs-Operationen gekoppelt ist, z. B. Bestimmen von Textur-Proben-Positionen (texture sample position), Lesen von Textur-Daten und Filtern der Textur-Daten. Textur-Daten werden von einem internen Textur-L1-Cache (nicht gezeigt) oder in einigen Ausführungsformen von dem L1-Cache innerhalb von SPM **310** gelesen und werden von einem L2-Cache, Parallel-Verarbeitungs-Speicher **204** oder System-Speicher **104** wie benötigt geholt. Jeder SPM **310** gibt verarbeitete Aufgaben an die Arbeits-Verteilungs-Kreuzschiene **330** aus, um die verarbeitete Aufgabe an einen anderen GPC **208** für weitere Verarbeitung bereitzustellen oder um die verarbeitete Aufgabe in einem L2-Cache, Parallel-Verarbeitungs-Speicher **204** oder System-Speicher **104** über Kreuzschiene-Einheit **210** zu speichern. Ein preROP (Vorraster-Operationen) **325** ist konfiguriert, um Daten von SPM **310** zu empfangen, Daten an ROP-Einheiten innerhalb der Partitions-Einheiten **215** zu richten, und Optimierungen für Farbmischung durchzuführen, Pixel-Farbdaten zu organisieren und Adress-Translationen durchzuführen.

[0043] Es wird geschätzt werden, dass die hierin beschriebene Kern-Architektur illustrativ ist und dass Variationen und Modifikationen möglich sind. Irgendeine Anzahl von Verarbeitungs-Einheiten, z. B. SPMs **310** oder Textur-Einheiten **315**, preROPs **325**, können innerhalb eines GPC **208** umfasst sein kann. Während nur ein GPC **208** gezeigt ist, kann eine PPU **202** irgendeine Anzahl von GPCs **208** umfassen, welche vorteilhafterweise funktionell ähnlich zueinander sind, so dass ein Ausführungs-Verhalten nicht davon abhängt, welcher GPC **208** eine bestimmte Verarbeitungs-Aufgabe empfängt. Ferner operiert jeder GPC **208** vorteilhafterweise unabhängig von anderen GPCs **208** unter Benutzung von separaten und distinkten Verarbeitungs-Einheiten L1-Caches, usw.

[0044] Fig. 3B ist ein Blockdiagramm einer Partitions-Einheit **215** innerhalb einer der PPUs **202** der **Fig.** 2B gemäß einer Ausführungsform der vorliegenden Erfindung. Wie gezeigt ist, umfasst die Partitions-Einheit **215** einen L2-Cache **350**, einen Frame-Puffer (FB), DRAM-Schnittstelle **355**, und eine Raster-Operations-Einheit (ROP) **360**. L2-Cache **350** ist ein Lese-/Schreibe-Cache, welcher konfiguriert ist, Lade- und Speicher-Operationen durchzuführen, welche von der Kreuzschiene-Einheit **215** und ROP **360** empfangen sind. Lese-Fehler (read misses) und dringende Rückschreibe-Anfragen (writeback requests) werden mittels L2-Cache **350** an FB-DRAM-Schnittstelle **355** zur Verarbeitung ausgegeben. Fehlerhafte oder schmutzige (dirty) Aktualisierungen werden auch an FB **355** für eine opportunistische Verarbeitung gesendet. FB **355** bildet direkt mit DRAM

220 eine Schnittstelle (Interfaces), wobei Lese- und Schreib-Anfragen ausgegeben werden und Daten, welche von DRAM **220** gelesen sind, empfangen werden.

[0045] In Grafik-Anwendungen ist ROP **360** eine Verarbeitungs-Einheit, welche Raster-Operationen durchführt, wie etwa Schablonieren (stencil), Z-Test, Mischung (blending), und dergleichen, und Pixel-Daten als verarbeitete Grafik-Daten zur Speicherung in Grafik-Speicher ausgibt. In einigen Ausführungsformen der vorliegenden Erfindung ist ROP **360** innerhalb jedes GPC **208** anstatt von Partitions-Einheit **215** umfasst, und Pixel-Lese- und Schreibe-Anfragen werden über Kreuzschiene-Einheit **210** anstatt von Pixel-Fragment-Daten übermittelt.

[0046] Die verarbeiteten Grafik-Daten können auf dem Anzeige-Gerät **110** angezeigt werden oder für weitere Verarbeitung mittels CPU **102** oder mittels irgendeiner der Verarbeitungs-Entitäten innerhalb des Parallel-Verarbeitungs-Subsystems **112** geleitet werden. Jede Partitions-Einheit **215** umfasst eine ROP **360**, um Verarbeitung der Raster-Operationen zu verteilen. In einigen Ausführungsformen kann ROP **360** konfiguriert sein, Z-Daten oder Farb-Daten zu komprimieren, welche an Speicher geschrieben sind, und Z-Daten oder Farb-Daten zu dekomprimieren, welche von Speicher gelesen werden.

[0047] Fachleute in der Technik werden verstehen, dass die in **Fig.** 2A, **Fig.** 2B, Fig. 3A und Fig. 3B beschriebene Architektur in keiner Weise den Geltungsbereich der vorliegenden Erfindung begrenzt und dass die hierin gelehrten Techniken auf irgendeiner korrekt konfigurierten Verarbeitungs-Einheit implementiert werden können, einschließlich ohne Begrenzung eine oder mehrere CPUs, eine oder mehrere Mehr-Kern-CPUs, eine oder mehrere PPUs **202**, ein oder mehrere GPCs **208**, eine oder mehrere Grafik- oder Spezialzweck-Verarbeitungs-Einheiten, oder dergleichen, ohne von dem Geltungsbereich der vorliegenden Erfindung abzuweichen.

[0048] In Ausführungsformen der vorliegenden Erfindung ist es wünschenswert, die PPU **202** oder andere Prozessor(en) eines Computer-Systems zu benutzen, um Allgemeinzweck-Berechnungen unter Benutzung von Thread-Feldern auszuführen. Jedem Thread in dem Thread-Feld ist ein eindeutiger Thread-Identifikator („Thread-ID") zugewiesen, welcher für den Thread während seiner Ausführung zugreifbar ist. Die Thread-ID, welche als ein eindimensionaler oder mehrdimensionaler numerischer Wert definiert werden kann, steuert verschiedene Aspekte des Verarbeitungs-Verhaltens des Threads. Zum Beispiel kann eine Thread-ID benutzt werden, um zu bestimmen, welchen Teil des Eingabe-Datensatzes ein Thread zu prozessieren hat, und/oder zu bestimmen, welchen Teil eines Ausgabe-Datensatzes ein Thread zu erzeugen hat oder zu schreiben hat.

[0049] Eine Sequenz von Pro-Thread-Anweisungen kann zumindest eine Anweisung umfassen, welche ein kooperatives Verhalten zwischen dem repräsentativen Thread und einem oder mehreren anderen Threads des Thread-Feldes definiert. Zum Beispiel könnte die Sequenz von Pro-Thread-Anweisungen eine Anweisung umfassen, um eine Ausführung von Operationen für den repräsentativen Thread bei einem bestimmten Punkt in der Sequenz anzuhalten (suspend), bis zu einer solchen Zeit, wenn einer oder mehrere der anderen Threads diesen bestimmten Punkt erreichen, eine Anweisung für den repräsentativen Thread, Daten in einem gemeinsamen Speicher zu speichern, auf welchen einer oder mehrere der anderen Threads zugreifen können, eine Anweisung für den repräsentativen Thread, um atomar Daten zu lesen und zu aktualisieren, welche in einem gemeinsamen Speicher gespeichert sind, auf welchen einer oder mehrere der anderen Threads Zugriff haben, basierend auf ihren Thread-IDs, oder dergleichen. Das CTA-Programm kann auch eine Anweisung umfassen, um eine Adresse in dem gemeinsamen Speicher zu berechnen, von welchem Daten zu lesen sind, wobei die Adresse eine Funktion der Thread-ID ist. Mittels eines Definierens von geeigneten Funktionen und mittels eines Bereitstellens von Synchronisations-Techniken können Daten auf eine bestimmte Stelle in dem gemeinsamen Speicher mittels eines Threads eines CTA geschrieben werden und von dieser Stelle mittels eines verschiedenen Threads desselben CTA in einer vorhersagbaren Weise gelesen werden. Folglich kann irgendein gewünschtes Muster von Daten-gemeinsam-Benutzen (data sharing) unter Threads unterstützt werden, und irgendein Thread in einem CTA kann mit irgendeinem anderen Thread in demselben CTA Daten gemeinsam nutzen bzw. teilen (share). Das Ausmaß, wenn überhaupt, eines gemeinsamen Benutzens von Daten unter Threads eines CTA ist mittels des CTA-Programms bestimmt; somit wird verstanden, dass in einer bestimmten Anwendung, welche CTAs benutzt, die Threads eines CTA tatsächlich Daten miteinander teilen bzw. benutzen könnten oder nicht, abhängig von dem CTA-Programm, und die Ausdrucke „CTA" und „Thread-Feld" werden hierin synonym benutzt.

[0050] Fig. 3C ist ein Blockdiagramm des SPM **310** von Fig. 3A gemäß einer Ausführungsform der vorliegenden Erfindung. Der SPM **310** umfasst einen Anweisungs-L1-Cache **370**, welcher konfiguriert ist, Anweisungen und Konstanten von Speicher über L1.5-Cache **335** zu empfangen. Eine Warp-Planer- (warp scheduler) und -Anweisungs-Einheit **312** empfängt Anweisungen und Konstanten von dem Anweisungs-L1-Cache **370** und

steuert eine lokale Register-Datei **304** und SPM **310** funktionale Einheiten gemäß den Anweisungen und Konstanten. Die SPM **310** funktionalen Einheiten umfassen N exec(Ausführung- oder Verarbeitung-)-Einheiten **302** und P Lade-Speicher-Einheiten (LSU) **303**.

[0051] SPM **310** stellt einen Auf-Chip (on-chip) (internen) Daten-Speicher mit verschiedenen Zugriffs-Niveaus bereit. Spezielle Register (nicht gezeigt) sind lesbar aber nicht mittels LSU **303** schreibbar und werden benutzt, um Parameter zu speichern, welche die „Position" jedes CTA-Threads definieren. In einer Ausführungsform umfassen spezielle Register ein Register pro CTA-Thread (oder pro exec-Einheit **302** innerhalb SPM **310**), welches eine Thread-ID speichert; jedes Thread-ID-Register ist nur mittels einer entsprechenden der exec-Einheit **302** zugreifbar. Spezielle Register können auch zusätzliche Register umfassen, welche mittels aller CTA-Threads lesbar sind (oder mittels aller LSUs **303**), welche einen CTA-Identifikator speichern, die CTA-Dimensionen, die Dimensionen eines Gitters (grid), zu welchem das CTA gehört, und einen Identifikator eines Gitters, zu welchem das CTA gehört. Spezielle Register werden während einer Initialisierung in Antwort auf Befehle geschrieben, welche über das Frontend **212** von dem Gerätetreiber **103** empfangen sind und welche sich während der CTA-Ausführung nicht ändern.

[0052] Ein Parameter-Speicher (nicht gezeigt) speichert Laufzeit-Parameter (Konstanten), welche gelesen werden können aber welche nicht mittels irgendeines CTA-Threads (oder irgendeiner LSU **303**) geschrieben werden können. In einer Ausführungsform stellt der Gerätetreiber **103** Parameter für den Parameter-Speicher bereit, bevor der SPM **310** darauf gerichtet wird, eine Ausführung eines CTA zu beginnen, welches diese Parameter benutzt. Irgendein CTA-Thread innerhalb irgendeines CTA (oder irgendeine exec-Einheit **302** innerhalb SPM **310**) kann auf globalen Speicher durch eine Speicher-Schnittstelle **214** zugreifen. Teile von globalem Speicher können in dem L1-Cache **320** gespeichert sein.

[0053] Die lokale Register-Datei **304** ist mittels jedes CTA-Threads als ein Notizzettel-Raum (scratch spase) benutzt; jedes Register wird für die exklusive Benutzung eines Threads alloziert und Daten in irgendeiner Register-Datei **304** sind nur von dem CTA-Thread zugreifbar, an welchen sie alloziert ist. Eine Ausnahme ist, wenn die lokale Register-Datei **304** einen oder mehrere skalare Register umfasst. Auf skalare Register kann mittels aller Threads in einer Thread-Gruppe zugegriffen werden, so dass mehrere Register nicht benötigt sind, um denselben skalaren Wert für jeden Thread in der Thread-Gruppe zu speichern. Die lokale Register-Datei **304** kann als eine Register-Datei implementiert sein, welche physikalisch oder logisch in P Spuren oder Bahnen aufgeteilt ist, wobei jede irgendeine Anzahl von Einträgen hat (wobei jeder Eintrag z. B. ein 32-Bit-Wort speichern könnte). Eine Spur ist jeder der N exec-Einheiten **302** und P Lade-Speicher-Einheiten LSU **303** zugewiesen und entsprechende Einträge in verschiedenen Spuren können mit Daten für verschiedene Threads populiert sein, welche dasselbe Programm ausführen, um eine SIMD-Ausführung zu ermöglichen. Verschiedene Teile der Spuren können an verschiedene der G gleichzeitigen Thread-Gruppen alloziert sein, so dass ein gegebener Eintrag in der lokalen Register-Datei **304** nur für einen bestimmten Thread zugreifbar ist. In einer Ausführungsform werden gewisse Einträge innerhalb der lokalen Register-Datei **304** zum Speichern von Thread-Identifikatoren reserviert, welche eines der speziellen Register implementieren.

[0054] Der gemeinsame Speicher **306** ist für alle CTA-Threads (innerhalb eines einzelnen CTA) zugreifbar; irgendeine Stelle in dem gemeinsamen Speicher **306** ist für irgendeinen CTA-Thread innerhalb desselben CTA zugreifbar (oder für irgendeine Verarbeitungs-Maschine innerhalb SPM **310**). Der gemeinsame Speicher **306** kann als eine gemeinsam benutzte oder geteilte (shared) Register-Datei oder ein gemeinsamer On-Chip-Cache-Speicher implementiert sein mit einer Zwischenverbindung, welche irgendeiner Verarbeitungs-Maschine erlaubt, von oder auf irgendeine Stelle in dem gemeinsamen Speicher zu lesen oder zu schreiben. In anderen Ausführungsformen könnte der gemeinsame Zustandsraum (shared state space) auf eine Pro-CTA-Region von Off-Chip-Speicher abbilden und könnte in L1-Cache **320** gecached sein. Der Parameter-Speicher kann als ein designierter Abschnitt innerhalb derselben gemeinsamen Register-Datei oder des gemeinsamen Cache-Speichers implementiert sein, welcher den gemeinsamen Speicher **306** implementiert, oder als eine separate gemeinsame Register-Datei oder ein On-Chip-Cache-Speicher, auf welchen die LSUs **303** nur-Lese-Zugriff haben. In einer Ausführungsform ist der Bereich, welcher den Parameter-Speicher implementiert, auch dazu benutzt, um die CTA-ID und die Gitter-ID zu speichern, sowie CTA- und Gitter-Dimensionen, wobei Teile der speziellen Register implementiert sind. Jede LSU **303** in SPM **310** ist mit einer unifizierten Adress-Abbildungs-Einheit **352** gekoppelt, welche eine Adresse, welche für Lade- und Speicher-Befehle bereitgestellt ist, welche in einem unifizierten Speicher-Raum spezifiziert ist, in eine Adresse in jedem distinkten Speicherraum zu konvertieren. Folglich kann eine Anweisung genutzt werden, um auf irgendwelche der lokalen, gemeinsamen oder globalen Speicherräume dadurch zuzutreiben, dass eine Adresse in dem unifizierten Speicherraum spezifiziert wird.

[0055] Der L1-Cache **320** in jedem SPM **310** kann benutzt werden, um private Pro-Thread-lokale-Daten und auch Pro-Applikation-globale-Daten zu cachen. In einigen Ausführungsformen können die Pro-CTA-geteilten Daten in dem L1-Cache **320** gecached werden. Die LSUs **303** sind mit einem uniformen L1-Cache **375**, dem gemeinsamen Speicher **306** und dem L1-Cache **320** über eine Speicher- und Cache-Zwischenverbindung **380** gekoppelt. Der uniforme Cache **375** ist konfiguriert, nur-Lese-Daten und Konstanten von Speicher über den L1.5-Cache **335** zu empfangen.

Temporäre SIMT-Ausführungs-Optimierung

[0056] Wie vorher beschrieben ist, stellen die GPUs ein bequemes und flexibles MIMD-Programmierungs-Modell auf einer SIMD-Hardware durch SIMT-Ausführung von Thread-Gruppen bereit. Die Threads innerhalb einer Thread-Gruppen können divergieren, weil der SPM **310** dynamisch einen Pro-Thread-Steuerfluss handhabt, wenn die Thread-Gruppen auf parallelen Ausführungs-Einheiten **302** ausführen. In einer räumlichen (spacial) SIMT-Architektur, wie in **Fig. 3C** gezeigt ist, kann die Anzahl von Ausführungs-Einheiten **302** gleich der Anzahl von Threads in einer Thread-Gruppe sein und Anweisungen werden in jedem Takt oder Zyklus (cycle) ausgeliefert. In einer temporären SIMT-Architektur sind die Threads in einer Thread-Gruppe zeitmultiplext (time-multiplexed) auf einer kleinen Anzahl von Ausführungs-Einheiten **302**. Zum Beispiel wird in einem SPM **310**, welcher vier parallele Ausführungs-Einheiten **302** hat und 32 Threads in einer Thread-Gruppe, die Anweisung für die Thread-Gruppe über acht aufeinander folgende Takte ausgeliefert.

[0057] Wenn eine räumliche SIMT-Architektur benutzt wird, ist eine Performanz typischerweise vermindert, wenn Threads in einer Thread-Gruppe divergieren, d. h. verschiedenen Steuerfluss-Pfaden folgen, und skalare Operationen werden redundant über Threads in der Thread-Gruppe ausgeführt, was zu einer Ineffizienz in der Performanz und Energieverbrauch führt. Temporäre SIMT-Ausführung kann benutzt werden, um Effektivitäten in Performanz und Energieverbrauch zu verbessern. Wenn die Threads in einer Thread-Gruppe temporär auf einer einzelnen exec-Einheit **302** anstatt räumlich auf parallelen exec-Einheiten **302** ausführen, kann die Anweisung einmal für die Thread-Gruppe ausgeführt werden anstatt einmal für jeden Thread, wenn die Anweisung als eine skalare Anweisung identifiziert ist. Wenn einige Threads in der Thread-Gruppe aufgrund von Divergenz inaktiv sind, werden Anweisungen nur zur Ausführung mittels der aktiven Threads in der Thread-Gruppe ausgeliefert. Divergenz tritt auf, wenn einer oder mehrere Threads in einer Thread-Gruppe Anweisungen ausführen, welche nicht mittels aller der Threads in der Thread-Gruppe ausgeführt werden. Divergenz kann von Wenn-Dann-Konstruktionen (if-when-constructs), Fall-Feststellungen (case statements), Sprung-Anweisungen und anderen konditionalen Operationen herrühren oder resultieren. In Summe wird Divergenz-Information und Skalar-Information benutzt, um eine temporäre Ausführung von parallelen Anweisungen dadurch zu optimieren, dass unnötige oder redundante Aktivität in exec-Einheiten **302** und in der lokalen Register-Datei **304** eliminiert werden.

[0058] **Fig. 4A** ist ein Blockdiagramm eines Teils des SPM **310** von **Fig. 3A**, welcher konfiguriert ist für temporäre SIMT-Ausführungen innerhalb einer einzelnen exec-Einheit **302**, gemäß einer Ausführungsform der vorliegenden Erfindung. Einer SIMT-Anweisung läuft durch verschiedene Pipeline-Stufen innerhalb von SPM **310**, wenn die Anweisung in einer temporären SIMT-Architektur ausgeführt wird. Die Warp-Zeitplaner- (warp scheduler) und Anweisungs-Einheit **312** umfasst eine Anweisungs-Frontend-Logik **410** und eine Anweisungs-Auslieferungs-Einheit **415**. Die Anweisungs-Frontend-Logik **410** empfängt einen Anweisungs-Strom und ist konfiguriert, in jedem Takt eine neue SIMT-Anweisung auszugeben. Wenn eine SIMT-Anweisung die Anweisungs-Auslieferungs-Einheit **415** erreicht, kann die SIMT-Anweisung temporär über einige Taktzyklen expandieren. Wie in der **Fig. 4A** gezeigt ist, wird eine einzelne Verarbeitungs-Pipeline benutzt, welche eine Ausführungs-(exec)-Einheit **402**, zwei Quell(SRC)-Operanden-Eingabe-Pipeline-Register **403** und **404**, und ein einzelnes Ziel-(destination)(DST)-Operanden-Ausgabe-Pipeline-Register **408** umfasst. Die Kettenplaner (warp scheduler) und Anweisungs-Einheit **312** liest Operanden, welche in Quell-Registern gespeichert sind, von der lokalen Register-Datei **304** und speichert temporär die Operanden in den SRC-Operanden-Eingabe-Pipeline-Registern **403** und **404**. Nach Prozessieren mittels der exec-Einheit **402**, wird das Ergebnis einer SIMT-Anweisung temporär in dem DST-Operanden-Ausgabe-Pipeline-Register **408** gespeichert. Die Kettenplaner- und Anweisungs-Einheit **312** transferiert dann die in dem DST-Operanden-Ausgabe-Pipeline-Register **308** gespeicherten Daten zudem Ziel-(destination)-Register in der lokalen Register-Datei **304** und speichert die Daten in dem Ziel-Register. In anderen Ausführungsformen werden ein oder mehrere zusätzliche oder weniger SRC-Operanden-Eingabe-Pipeline-Register benutzt und es werden ein oder mehrere zusätzliche DST-Operanden-Ausgabe-Pipeline-Register benutzt.

[0059] Wenn eine einzelne exec-Einheit **402** in der temporären SIMT-Architektur benutzt wird, kann die SIMT-Anweisung einmal für jeden Thread in der Thread-Gruppe ausgeliefert werden, wenn alle der Threads die An-

weisung ausführen müssen. In einigen Umständen kann die SIMT-Anweisung für einen einzelnen Takt oder Zyklus ausgeliefert sein, in mehreren Zyklen für nur einen Teil der Threads oder überhaupt nicht. In einer temporären SIMT-Architektur, welche vier Parallel-Verarbeitungs-Pipelines und 32 Threads in einer Thread-Gruppe hat, wird eine SIMT-Anweisung über acht aufeinander folgende Zyklen ausgeliefert, wenn alle der Threads die SIMT-Anweisung ausführen müssen. Für jeden Takt oder Zyklus beginnt eine neue Auslieferungs-Untergruppe von Threads eine Ausführung in paralleler Weise; Threads 0 bis 3 in dem ersten Auslieferungs-Zyklus, Threads 4 bis 7 in dem zweiten Auslieferungs-Zyklus, etc. Wenn in einigen Umständen nicht alle der Threads in einer Auslieferungs-Untergruppe die SIMT-Anweisung auszuführen brauchen, dann wird die SIMT-Anweisung nicht für diese Auslieferungs-Untergruppe ausgeliefert. Verglichen mit der räumlichen SIMT-Architektur werden die exec-Einheiten **302** und **402** effizienter benutzt.

[0060] Konventionelle temporäre Ausführungs-Architekturen implementieren eine rigide Zeitplanung oder Planung (scheduling) von Operationen. Zum Beispiel liefert eine konventionelle temporäre Ausführungs-Architektur immer jede SIMT-Anweisung für jeden Thread in der Thread-Gruppe aus. Ferner werden Steuersignale derart erzeugt, dass alle Threads einen ähnlichen Satz von Operationen durchführen, wenn sie die Pipeline-Stufen traversieren. Zum Beispiel lesen alle Threads in einer Thread-Gruppe Quell-Operanden von einer Register-Datei, führen die spezifizierte arithmetische Operation in Ausführungs-Einheiten durch und schreiben Ergebnisse zurück in die Register-Datei.

[0061] Im Gegensatz dazu ist die Anweisungs-Auslieferungs-Einheit **415** optimiert, unnötige oder redundante Operationen zu eliminieren. Die Optimierung kann eine Energieverminderung durch Eliminieren von Operationen umfassen, wie etwa wenn die lokale Register-Datei **304** liest und schreibt. Wenn eine exec-Einheit **302** parallel zu zumindest einer anderen exec-Einheit **302** ist, welche für einen Thread-Takt (thread clocking) von Pipeline-Registern untätig ist, kann die untätige exec-Einheit **302** eliminiert werden, um einen Energieverbrauch zu vermindern. Die Anweisungs-Auslieferungs-Einheit **415** kann auch konfiguriert sein, Auslieferungs-Zyklen von dem Ausführungs-Plan (execution schedule) zu eliminieren, um den Anweisungs-Durchsatz für einen oder mehrere Threads in einer Thread-Gruppe zu verbessern.

[0062] Ausführungs-Divergenz tritt auf, wenn die Threads in einer Thread-Gruppe unabhängig verzweigen und verschiedenen Steuerflusspfaden folgen. Um die Divergenz zu handhaben, hält die Kettenplaner- und Anweisungs-Einheit **312** einen Stapel von Kettenprogramm-Zählern (warp program counters) aufrecht, zusammen mit aktiven Masken, welche anzeigen, welcher Thread in der Thread-Gruppe die Ziel-Anweisung ausführen sollte. Eine divergenter-Zweig-Anweisung führt zu zwei Programm-Zählern und zu zwei aktiven Masken für die genommenen bzw. nicht genommenen Pfade und ein Satz wird auf den Stapel geschoben (pushed), während Threads-Anweisungen unter Benutzung des anderen Satzes von Programm-Zähler und aktiver Maske ausführen. Die Kettenplaner- und Anweisungs-Einheit **312** führt Ketten (warps) mit einem Programmzähler und einer aktiven Maske zu einer Zeit aus und handhabt Divergenz und erneute Konvergenz (re-convergence) dadurch, dass Programm-Zähler und aktive Masken von dem Stapel geschoben und entnommen (pushing and popping) werden.

[0063] Wenn eine divergente Kette auf konventionellen räumlichen und temporären SIMT-Architekturen ausführt, werden Anweisungen sogar für inaktive Threads ausgeliefert, wodurch eine Verarbeitungs-Performanz reduziert ist. Wenn z. B. nur vier Threads in einer Thread-Gruppe von 32 Threads aktiv sind, vermindert sich die Benutzung der Verarbeitungs-Pipeline von 100% auf 12,5%. Divergenz ist ein Haupt-Performanz-Fallstrick für Mehr-Thread-Prozessoren und Programmierer müssen sorgfältig sein, um die Threads kohärent zu halten, um Durchsatz zu optimieren.

[0064] Um Effizienz beim Vorhandensein von Ausführungs-Divergenz zu optimieren, benutzt die Anweisungs-Auslieferungs-Einheit **415** die aktive Maske für die Thread-Gruppe, um zu steuern oder zu kontrollieren, welche Threads jede SIMT-Anweisung ausführen. Anstatt eine SIMT-Anweisung über eine fixe oder feste Anzahl von Zyklen zur Ausführung mittels aller der Threads in einer Thread-Gruppe auszuliefern, kann die Anweisungs-Auslieferungs-Einheit **415** eine SIMT-Anweisung über weniger Zyklen ausliefern, wenn einige Threads in der Thread-Gruppe inaktiv sind. In einer Ausführungsform ist die Anweisungs-Auslieferungs-Einheit **415** konfiguriert, nicht die SIMT-Anweisung für Auslieferungs-Untergruppen von sequentiellen Threads auszuliefern, welche gemäß der aktiven Maske inaktiv sind. Wenn die lokale Register-Datei **304** unabhängige Indizierung (indexing) pro Thread unterstützt, wählt eine andere Ausführungsform nicht sequentielle aktive Threads zum Ausliefern in jedem Zyklus. Als eine weitere Optimierung kann die Anweisungs-Auslieferungs-Einheit **415** auch Prädikations-Register-Werte und ein Anweisungs-Prädikations-Feld beim Bestimmen von Threads, welche die SIMT-Anweisung nicht ausführen, berücksichtigen und kann weiter die Anzahl von Zyklen vermindern, für welche die SIMT-Anweisung ausgeliefert wird.

[0065] Tabelle 1 zeigt eine Beispiel-Ausführungs-Zeitlinie, in welcher unnötige Auslieferungs-Zyklen eliminiert sind, was einem SPM **310** erlaubt, mehr SIMT-Anweisungen in einer gegebenen Zeitperiode auszuführen. Die Steuerlogik-Optimierungen können auf verschiedenen Eingaben basieren, wie etwa Divergenz-Information, z. B. einer aktive Maske für die Thread-Gruppe, Prädikations-Register-Werte, oder Skalar-Information, z. B. Anweisungs-Flaggen, bestimmten Opcodes, und Register-Spezifikatoren. Die aktive Maske zeigt an, welche Threads in einer Thread-Gruppe aktiv sind und die SIMT-Anweisung ausführen sollten. Das in Tabelle 1 gezeigte Beispiel ist ein SPM, welche konfiguriert ist, eine SIMT-Anweisung zur Ausführung mittels eines einzelnen Threads in jedem Zyklus auszuliefern, wobei acht Threads in einer Thread-Gruppe sind.

Tabelle 1 – SIMT-Anweisungs-Auslieferungs-Zeitlinie

Eingabe-Strom	Eingabe an Auslieferung	Auslieferungs-Ausgabe
I1		
I2	I1	
I3	I2	I1 Thread1
	-	I1 Thread3
	-	I1 Thread4
I4	I3	I2 Thread1
-	I4	I3 Thread1
-	-	I3 Thread2
-	-	I3 Thread3
-	-	I3 Thread4
-	-	I3 Thread5
-	-	I3 Thread6
-	-	I3 Thread7
	-	I3 Thread8
		I4 Thread2
		I4 Thread5
		I4 Thread6
		I4 Thread7
		I4 Thread8

[0066] Basierend auf der aktiven Maske gibt die Anweisungs-Auslieferungs-Einheit **415** die erste SIMT-Anweisung aus, I1 wird für Thread 1, Thread 3 und Thread 4 ausgegeben. Die Anweisungs-Frontend-Logik **410** kommt zum Stillstand (stalls) und kann keine neue Eingabe-Anweisung annehmen, während die Anweisungs-Auslieferungs-Einheit **415** die erste SIMT-Anweisung für mehrere Zyklen ausgibt.

[0067] In einer SIMT-Architektur führen Threads oft redundant eine Anweisung aus, welche eine skalare Buchhaltungs-Operation (book keeping operation) durchführt oder welche gemeinsame skalare Operanden empfängt. Beispiele umfassen gemeinsame Steuerfluss-Anweisungen wie das Zähler-Inkrement und konditionale Verzweigung für eine gezählte Schleife (counted loop). Andere Beispiele umfassen Speicher-Adress-Berechnungen, wie etwa ein Berechnen von Basis-Adressen, wenn die Threads in einer Thread-Gruppe auf Vektoren und Matrizen operieren.

[0068] Die zweite Anweisung, I2, gezeigt in Tabelle 1, ist eine skalare SIMT-Anweisung, was bedeutet, dass alle der Threads in der Thread-Gruppe eine Anweisung ausführen würden, welche dasselbe Ergebnis erzeugen wird. Daher wird die zweite Anweisung nur für einen Thread (Thread 1) im Interesse von allen der aktiven Threads in der Thread-Gruppe ausgeführt. Die Anweisungs-Auslieferungs-Einheit **415** bestimmt, dass die dritte Anweisung, I3 mittels der acht Threads in der Thread-Gruppe ausgeführt werden sollte. Daher wird die SIMT-Anweisung für jeden der acht Threads ausgegeben. Die vierte Anweisung, I4 wird an fünf der acht Threads basierend auf der aktiven Maske ausgegeben. Die Anweisungs-Frontend-Logik **410** geht in den Ruhe-Zustand

(stalls) und kann keine neue Eingabe-Anweisung annehmen, während die Anweisungs-Auslieferungs-Einheit **415** die dritte und die vierte SIMT-Anweisung für mehrere Zyklen ausgibt.

[0069] Tabelle 1 demonstriert, wie eine SIMT-Anweisung mittels einer Anweisungs-Auslieferungs-Einheit **415** für eine variable Anzahl von Zyklen ausgegeben werden kann. Die Anweisungs-Frontend-Logik **410** sollte ausgebildet sein, diese Variabilität zu berücksichtigen, z. B. dadurch, dass sie in den Ruhezustand versetzt, bis die Anweisungs-Auslieferungs-Einheit **415** bereit ist, eine neue Anweisung anzunehmen. Als eine Alternative zu einem in-den-Ruhezustand-Versetzen der frühen Stufen des SPM **310** können Queues benutzt werden, um verschiedene Verarbeitungs-Einheiten zu entkoppeln. Zum Beispiel kann eine kleine Queue zwischen der Anweisungs-Frontend-Logik **410** und der Anweisungs-Auslieferungs-Einheit **415** lokalisiert sein.

[0070] Fig. 4B illustriert das Format einer SIMT-Anweisung **440** gemäß einer Ausführungsform der Erfindung. Um Optimierungen zu ermöglichen, welche eine Skalarisierung betreffen, gibt es verschiedene Wege, in welchen Operanden oder Operationen für die Threads in einer Thread-Gruppe als skalar identifiziert werden können. Ein Compiler kann gleichförmige Anweisungen und/oder gleichförmige Operanden (auch Skalare genannt) identifizieren und kann skalare Information durch Flaggen (flags) entweder in oder assoziiert mit einer Anweisung kommunizieren. Ein oder mehrere Register können definiert werden (indiziert mit gewissen Register-Spezifikatoren), welche logisch oder physikalisch von den Threads in einer Thread-Gruppe gemeinsam benutzt oder geteilt werden (shared) und welche als skalare Register definiert sind. Ähnlich können ein oder mehrere Anweisungs-Opcodes als skalare Operationen definiert werden, welche zu skalaren Anweisungen führen. In anderen Fällen kann die Ketten-Planer- und Anweisungs-Einheit **312** dynamisch skalare Operanden und Operationen (oder Anweisungen) nachverfolgen oder nachvollziehen (track). Zum Beispiel kann die Ketten-Planer- und Anweisungs-Einheit **312** Flaggen setzen und propagieren, um gleichförmige (oder skalare) Register zu identifizieren. Die Ketten-Planer- und Anweisungs-Einheit **312** kann die Flaggen basierend auf einer Ausführung von gewissen Anweisungen setzen, welche definiert sind, um skalare Ergebnisse zu erzeugen (wie etwa eine Last-uniforme-Anweisung). In anderen Implementierungen kann die Ketten-Planer- und Anweisungs-Einheit **312** tatsächliche Daten-Werte, welche in den Registern gespeichert sind, welche mittels der Anweisungs-Operanden spezifiziert sind, vergleichen, um zu bestimmen, ob der Operand und/oder die Anweisung skalar ist.

[0071] Wie in Fig. 4B gezeigt ist, umfasst die SIMT-Anweisung **440** eine skalarer-Opcode-Flagge **445**, welche anzeigt, ob ein Opcode **441** skalar ist. Wenn der Opcode **441** skalar ist, ist auch die SIMT-Anweisung **440** skalar. Die skalare-Operand-Flaggen **446** zeigen an, ob jeder des DST-Operanden **448**, des SCR-Operanden **443** bzw. des SCR-Operanden **444** skalar ist. Wenn der DST-Operand **448** skalar ist, ist die SIMT-Anweisung **440** auch skalar. Wenn es mehrere DST-Operanden gibt, müssen alle der DST-Operanden skalar sein, damit die SIMT-Anweisung **440** eine skalare Anweisung ist. In einer anderen Ausführungsform können spezifische Opcodes und/oder SCR- oder DST-Operanden als skalar vordefiniert sein, so dass die skalarer-Opcode-Flaggen **445** und/oder skalarer-Operand-Flaggen **446** nicht benötigt werden.

[0072] In einer Ausführungsform wird eine separate Skalar-Einheit benutzt, um skalare Operationen durchzuführen, um die Effizienz in einer SIMT-Architektur zu optimieren. Eine Skalar-Register-Datei kann konfiguriert sein, um skalare Operanden zu speichern und dedizierte Skalar-Ausführungs-Einheiten können konfiguriert sein, skalare Operationen durchzuführen. Ein Nachteil dieses Zugangs ist, dass die Skalar-Register-Datei und Skalar-Einheit zusätzliche Schaltungen sind, welche nur für Skalar-Verarbeitung vorgesehen ist (dedicated). Weiterhin müssen die skalaren Operanden zu den Verarbeitungs-Pipelines geleitet werden, um als Quellen für die regulären parallelen SIMT-Anweisungen benutzt zu werden. In einer anderen Ausführungsformen werden skalare Operationen in den exec-Einheiten **402** und **302** durchgeführt, skalare Operanden werden in der lokalen Register-Datei **304** gespeichert und eine skalare Ausführung ist temporär anstatt räumlich optimiert.

[0073] Die Anweisungs-Auslieferungs-Einheit **415** benutzt Information über skalare Operanden und skalare Operationen zusätzlich zu der Maske, um unnötige oder redundante Verarbeitung oder Register-Lese- und Schreib-Operationen zu eliminieren. Wenn irgendwelche Quellen-Operanden **443** oder **444** als skalar identifiziert sind, müssen die skalaren SRC-Operanden nicht von der lokalen Register-Datei **304** in jedem Zyklus gelesen werden, in dem die SIMT-Anweisung **440** ausgeliefert wird. Skalare SRC-Operanden können in die SRC-Pipeline-Register **403** und/oder **404** einmalig transferiert werden, wenn die SIMT-Anweisung **440** für den nächsten Thread oder Auslieferungs-Untergruppe ausgeliefert wird, und in SRC-Registern **403** und **404** zur Benutzung gespeichert werden, wenn die SIMT-Anweisung **440** während nachfolgender Zyklen ausgeliefert wird. Ein Vermindern der Anzahl von Malen, für die auf die SRC-Register zugegriffen wird, vermindert die Energie, welche verbraucht wird, um die SRC-Operanden von der Register-Datei zu lesen und um die SRC-Operanden in die SRC-Pipeline-Register zu schreiben.

[0074] Wenn ferner alle Quell-Operanden **443** und **444** skalar sind, ist es nicht notwendig, Operationen in der exec-Einheit **402** für mehrere Threads in der Thread-Gruppe redundant durchzuführen. Nachdem der erste Thread oder Threads in einer Auslieferungs-Untergruppe die SIMT-Anweisung ausführen, können die Ergebnisse in dem DST-Register **408** oder Registern innerhalb der exec-Einheiten **302** oder **402** für nachfolgende Threads oder Threads in einer Auslieferungs-Untergruppe gehalten werden, um in die lokale Register-Datei **304** oder in eine andere Destination zu schreiben. Noch ferner, wenn die SIMT-Anweisung **440** als skalar identifiziert ist, führen nur ein einzelner Thread oder Threads in einer Auslieferungs-Untergruppe die skalare SIMT-Anweisung aus und schreiben in die lokale Register-Datei **304** zurück oder in eine andere Destination. Diese Optimierungen, welche die Anzahl von Zyklen vermindern, für welche eine SIMT-Anweisung ausgeliefert ist, basierend auf skalaren Opcodes und/oder Anweisungen, verbessern einen Anweisungs-Durchsatz und vermindern einen Energie-Verbrauch, wobei die Optimierungen basierend auf skalaren Operanden primär einen Energie-Verbrauch vermindern, weil weniger Register-Lese- und Schreib-Operationen durchgeführt werden.

[0075] Die Ausführung von Speicher-Zugriffen kann unter Benutzung von skalaren Operanden optimiert werden. In konventionellen Mehr-Thread-Prozessoren erzeugen Threads Speicher-Adressen unabhängig und eine Speicher-Schnittstelle versucht, die Adressen in breite Blockzugriffe auf das Speichersystem zu koaleszieren. Zum Beispiel können 32 individuelle 4-Byte-Wort-Zugriffe in eine 128-Byte-Anfrage koalesziert werden. Skalare Operanden können mittels Lade- und Speicher-Anweisungen benutzt werden, um explizit strukturierte Speicher-Zugriffs-Muster zu kodieren, welche individuelle Thread-Adressen als eine Funktion einer skalaren Basis-Adresse summiert mit der Thread-ID des Threads multipliziert mit einem gleichförmigen Schritt (uniform stride) erzeugen. Die unifizierte-Adresse-Abbildungs-Einheit **352** kann dann direkt breite Block-Zugriffe auf das Speichersystem basierend auf dem skalaren Operanden erzeugen, ohne zuerst individuelle Thread-Adressen erzeugen zu müssen, nur um sie zurück zusammen zu koaleszieren. Eliminieren einer Adress-Erzeugung und Koaleszieren kann signifikante Effienz-Einsparungen für viele gewöhnliche Speicher-Zugriffsmuster bieten.

[0076] Soweit sind temporäre Ausführungs-Optimierungen für Operanden und Operationen beschrieben worden, welche uniform oder gleichmäßig über eine Thread-Gruppe sind, aber die Optimierungs-Techniken sind in gleicher Weise anwendbar, wenn es skalare Operanden und Operationen über mehrere Thread-Gruppen gibt (z. B. diese in einem CTA). Im Allgemeinen muss eine SIMT-Architektur Skalare für jede Thread-Gruppe replizieren, da Thread-Gruppen unabhängig ausführen und anderenfalls es Lese-/Schreib-Gefahren (hazards) geben kann. In gewissen Quellen, wie etwa wenn Daten nur lesbar sind für einen Kernel oder wenn mehrere Thread-Gruppen synchronisiert sind, können jedoch skalare Operanden von mehreren Thread-Gruppen gemeinsam benutzt oder geteilt werden (shared).

[0077] Die Optimierungs-Techniken sind auch anwendbar, wenn Operanden nur für die Threads skalar sind, welche mittels derselben exec-Einheit **302** ausgeführt werden, wenn mehrere exec-Einheiten **302** in dem SPM **310** benutzt werden, anstatt skalar über eine gesamte Thread-Gruppe. Insbesondere kann ein skalarer Reduktions-Operand für Threads in einer Thread-Gruppe identifiziert werden, welche mittels derselben exec-Einheit **302** verarbeitet werden, wenn Reduktions-Operationen über die Threads durchgeführt werden, welche mittels derselben exec-Einheit **302** verarbeitet werden. Wenn eine Reduktions-Operation durchgeführt wird, ist ein DST-Operand skalar und einer oder mehrere SRC-Operanden sind nicht skalar. Das Ergebnis der Reduktions-Operation, welches in dem DST-Ausgabe-Pipeline-Register **408** gespeichert ist, muss nur einmal in das DST-Operanden-Register in der lokalen Register-Datei **304** oder in einer andere Destination für alle der aktiven Threads der Thread-Gruppe geschrieben werden, welche mittels derselben exec-Einheit **302** verarbeitet werden, um dadurch eine Energie zu reduzieren, welche verbraucht wird, verglichen mit einem Schreiben der lokalen Register-Datei **304** für jeden aktiven Thread in der Thread-Gruppe.

[0078] In einer Ausführungsform ist eine dedizierte Skalar-Schaltung zum Durchführen von skalaren Operationen innerhalb des SPM **310** umfasst und skalare Anweisungen werden zur Ausführung mittels der dedizierten Skalar-Schaltung ausgeliefert. Eingaben für die dedizierte Skalar-Schaltung können mittels des DST-Ausgabe-Pipeline-Registers **408** oder der lokalen Register-Datei **304** bereitgestellt sein. Ähnlich können Ausgaben von der dedizierten Skalar-Schaltung in die SRC-Eingabe-Pipeline-Register **403** und/oder **404** oder die lokale Register-Datei **304** gespeichert sein. Ein Compiler kann konfiguriert sein, zwischen skalaren Operanden, welche SRC-Operanden für skalare SIMT-Anweisungen sind, und skalaren Operanden, welche SRC-Operanden für nicht-skalare SIMT-Anweisungen sind, unterscheiden und diese Information kann von der Anweisungs-Auslieferungs-Einheit **415** benutzt werden, um die SIMT-Anweisungen zur Ausführung mittels der dedizierten Skalar-Schaltung zeitlich zu planen (schedule).

[0079] Fig. 5A ist ein Flussdiagramm von Verfahrensschritten zum Ausliefern (dispatching) einer Anweisung zur Ausführung mittels einer Thread-Gruppe gemäß einer Ausführungsform der vorliegenden Erfindung. Ob-

wohl die Verfahrensschritte in Verbindung mit dem Systemen von Fig. 1, Fig. 2, Fig. 3A, Fig. 3B, Fig. 3C und Fig. 4A beschrieben sind, werden Fachleute in der Technik verstehen, dass irgendein System, welches konfiguriert ist, die Verfahrensschritte auszuführen, in irgendeiner Ordnung, innerhalb des Geltungsbereichs der Erfindung ist. Bei Schritt **505** wird eine SIMT-Anweisung zur Ausführung mittels paralleler Threads einer Thread-Gruppe mittels der Anweisungs-Auslieferungs-Einheit **415** empfangen. Bei Schritt **510** evaluiert die Anweisungs-Auslieferungs-Einheit **415** die Anweisung, um eine skalare Charakteristik zu identifizieren, z. B. skalare Operanden und/oder Opcodes, wie in größerem Detail in Verbindung mit Fig. 5B beschrieben ist.

[0080] Bei Schritt **515** bestimmt die Anweisungs-Auslieferungs-Einheit **415**, ob keiner der Threads in der Thread-Gruppe aktiv ist und, wenn dem so ist, schreitet die Anweisungs-Auslieferungs-Einheit **415** direkt zu Schritt **540** vor, ohne die SIMT-Anweisung auszuliefern. Wenn bei Schritt **515** die Anweisungs-Auslieferungs-Einheit **415** bestimmt, dass zumindest einer der Threads in der Thread-Gruppe aktiv ist, dann bestimmt bei Schritt **520** die Anweisungs-Auslieferungs-Einheit **415**, ob die SIMT-Anweisung eine skalare Anweisung ist. Wenn die SIMT-Anweisung eines skalare Anweisung ist, dann liefert bei Schritt **522** die Anweisungs-Auslieferungs-Einheit **415** die SIMT-Anweisung einmalig für den ersten aktiven Thread aus und schreibt das DST-Register auf die lokale Register-Datei **304** nur für den ersten aktiven Thread. Bei Schritt **540** ist die Auslieferung der SIMT-Anweisung vollendet.

[0081] Wenn bei Schritt **520** die Anweisungs-Auslieferungs-Einheit **415** bestimmt, dass die SIMT-Anweisung nicht eine skalare Anweisung ist, dann bestimmt bei Schritt **525** die Anweisungs-Auslieferungs-Einheit **415**, ob die SIMT-Anweisung alle skalaren SRC-Operanden umfasst. Wenn die SIMT-Anweisung alle skalaren SRC-Operanden umfasst, dann liefert bei Schritt **527** die Anweisungs-Auslieferungs-Einheit **415** die SIMT-Anweisung nur für den ersten aktiven Thread aus, wobei die SRC-Register von der lokalen Register-Datei **304** nur für den ersten aktiven Thread ausgegeben werden, wobei eine Operation in der exec-Einheit **402** nur für den ersten aktiven Thread durchgeführt wird und wobei das DST-Pipeline-Register **408** nur für den ersten aktiven Thread geschrieben wird. Die Anweisungs-Auslieferungs-Einheit **415** schreibt dann das DST-Register auf die lokale Register-Datei **304** für jeden der aktiven Threads. Bei Schritt **540** ist die Auslieferung (dispatch) der SIMT-Anweisung vollendet.

[0082] Wenn bei Schritt **530** die Anweisungs-Auslieferungs-Einheit **415** bestimmt, dass die SIMT-Anweisung nicht zumindest einen skalaren SRC-Operanden umfasst, dann liefert bei Schritt **537** die Anweisungs-Auslieferungs-Einheit **415** die SIMT-Anweisung für alle aktiven Threads aus und die SRC-Register werden für alle der aktiven Threads geladen. Wenn die SIMT-Anweisung nicht zumindest einen skalaren SRC-Operanden umfasst, dann liefert bei Schritt **532** die Anweisungs-Auslieferungs-Einheit **415** die SIMT-Anweisung aus, wobei die SRC-Register von der lokalen Register-Datei **304** nur für den ersten aktiven Thread in der Thread-Gruppe gelesen werden. Bei Schritt **535** liefert die Anweisungs-Auslieferungs-Einheit **415** die SIMT-Anweisung für die anderen aktiven Threads aus, ohne die skalaren SRC-Register erneut zu lesen. Der skalare Operand wird nur einmal in das SRC-Register geladen, um dadurch einen Energieverbrauch zu reduzieren, welcher von einem Speicher-Zugriff oder Register-Datei-Zugriff herrührt, um den SRC-Operanden zu dem SRC-Register zu transferieren. Nach Schritten **535** und **537** schreitet die Anweisungs-Auslieferungs-Einheit **415** zu Schritt **540** vor und die Auslieferung der SIMT-Anweisung ist vollendet.

[0083] Fig. 5B ist ein Flussdiagramm von Verfahrensschritten für Schritte **510**, welche in Fig. 5A gezeigt ist, gemäß einer Ausführungsform der vorliegenden Erfindung. Bei Schritt **550** bestimmt die Anweisungs-Auslieferungs-Einheit **415**, ob der Opcode der SIMT-Anweisung ein skalarer Opcode ist, und, wenn dem so ist, identifiziert bei Schritt **565** die Anweisungs-Auslieferungs-Einheit **415** die SIMT-Anweisung als eine skalare Anweisung. Anderenfalls bestimmt bei Schritt **555** die Anweisungs-Auslieferungs-Einheit **415**, ob alle der DST-Operanden skalar sind, und, wenn dem so ist, identifiziert bei Schritt **565** die Anweisungs-Auslieferungs-Einheit **415** die SIMT-Anweisung als eine skalare Anweisung. Anderenfalls bestimmt bei Schritt **560** die Anweisungs-Auslieferungs-Einheit **415**, ob alle der SRC-Operanden Skalare sind, und, wenn dem so ist, identifiziert bei Schritt **568** die Anweisungs-Auslieferungs-Einheit **415**, dass alle der SRC-Operanden skalare Operanden sind und dass die SIMT-Anweisung nicht eine skalare Anweisung ist. Anderenfalls bestimmt bei Schritt **570** die Anweisungs-Auslieferungs-Einheit **415**, ob irgendwelche/jegliche (any) der SRC-Operanden skalar sind, und, wenn dem so ist, identifiziert bei Schritt **575** die Anweisungs-Auslieferungs-Einheit **415** jeden skalaren SRC-Operanden als einen skalaren Operanden. Wenn die Anweisungs-Auslieferungs-Einheit **415** bei Schritt **570** bestimmt, dass keiner der SRC-Operanden skalar ist, dann hat bei Schritt **580** die Anweisungs-Auslieferungs-Einheit identifiziert, dass die SIMT-Anweisung nicht eine skalare Anweisung ist und dass die SIMT-Anweisung keinen einzigen skalaren Operanden umfasst. Die Anweisungs-Auslieferungs-Einheit **415** kehrt dann zu Schritt **515** zurück.

[0084] Tabelle 2 fasst die Optimierungen zusammen, welche basierend auf einer Identifikation von skalaren Operanden und skalaren Anweisungen eingesetzt werden können, wie vorher im Zusammenhang mit Fig. 5A und Fig. 5B beschrieben ist. „Einmalig" zeigt an, dass die SIMT-Anweisung an exec-Einheit **402** einmalig ausgeliefert wird oder dass die lokale Register-Datei **304** einmalig gelesen oder geschrieben wird. „N" zeigt an, dass die SIMT-Anweisung für jeden aktiven Thread ausgeliefert wird oder dass die lokale Register-Datei **304** für jeden aktiven Thread gelesen wird oder geschrieben wird.

Tabelle 2

Eingaben				Auslieferungs-Steuerung		
Skalare Anweisung	Alle skalare DST-Operanden	Alle skalare SRC-Operands	1+ skalare SRC-Operanden	Auslieferung an exec-Einheit	Lese-SRC-Register	Schreib-DST-Register
ja	X	X	X	einmalig	einmalig	einmalig
X	Ja	ja	ja	einmalig	einmalig	einmalig
nein	Nein	ja	ja	einmalig	einmalig	N
nein	nein	nein	ja	N	einmalig/N	N
nein	ja	nein	ja	N	einmalig/N	einmalig

[0085] Temporäre SIMT-Optimierungen können zunehmend wichtig werden, weil physikalische Begrenzungen Schaltungs-Designs zu kleineren Kernen treiben, z. B. SPMs **310** oder GPCs **208**, mit weniger Ausführungs-Einheiten und mehr lokalisierte-Steuerung- und Daten-Pfaden. Die temporären SIMT-Optimierungen können die Anzahl von Malen vermindern, in der eine SIMT-Anweisung ausgeliefert wird, oder kann die Anzahl von SRC-Register-Lese-Operationen oder DST-Register-Schreib-Operationen vermindern, um dadurch einen Energieverbrauch zu vermindern. Insgesamt ist eine Benutzung oder Verarbeitungs-Ressourcen verbessert und die Performanz kann auch erhöht werden, wenn temporäre SIMT-Optimierungen eingesetzt werden.

[0086] Eine Ausführungsform der Erfindung kann als ein Programm-Produkt zur Benutzung mit einer Computer-System implementiert sein. Das Programm oder die Programme des Programm-Produkts definieren Funktionen der Ausführungsformen (einschließlich der hierin beschriebenen Verfahren) und können auf einer Verschiedenheit von Computer-lesbaren Speichermedien beinhaltet sein. Illustrative Computer-lesbare Speichermedien umfassen, sind jedoch nicht darauf beschränkt: (i) nicht-schreibbare Speichermedien, z. B. Nur-Lese-Speicher-Geräte innerhalb eines Computers (wie CD-ROM-Platten, welche mittels eines CD-ROM-Laufwerks lesbar sind, Flash-Speicher, ROM-Chips oder irgendein anderer Typ von Festkörper-nicht-volatilem Halbleiter-Speicher), auf welchen Informationen permanent gespeichert ist; und (ii) schreibbare Speichermedien (z. B. Floppy-Disks innerhalb eines Disketten-Laufwerks oder eines Festplatten-Laufwerks oder irgendein anderer Typ von Festkörper-Halbleiter-Speicher mit willkürlichem Zugriff), auf welchen veränderbare Informationen gespeichert ist.

[0087] Die Erfindung ist oben mit Bezug auf spezifische Ausführungsformen beschrieben worden. Fachleute in der Technik werden jedoch verstehen, dass verschiedene Modifikationen und Änderungen daran gemacht werden können, ohne von dem weiteren Geist und Geltungsbereich abzuweichen, wie in den angehängten Ansprüchen ausgeführt. Die vorangehende Beschreibung und die Zeichnungen sind demgemäß in einem illustrativen anstatt in einem restriktiven Sinne anzusehen.

Patentansprüche

1. System zum Ausführen von Anweisungen, wobei das System aufweist:
einen Speicher, welcher konfiguriert ist, Anweisungen zur Ausführung mittels Threads zu speichern; und
einen Einzel-Anweisung-Mehr-Thread(SIMT)-Prozessor, welcher konfiguriert ist, um:
eine Anweisung zur Ausführung mittels Threads in einer Thread-Gruppe zu empfangen;
die Anweisung zu evaluieren, um eine skalare Charakteristik zu identifizieren, und
die Anweisung zur Ausführung mittels eines Teils der Threads in der Thread-Gruppe basierend auf der skalaren Charakteristik ausliefern.

2. System gemäß Anspruch 1, wobei der SIMT-Prozessor ferner konfiguriert ist, die Anweisung nur für einen ersten Thread in der Thread-Gruppe auszuliefern, wenn die skalare Charakteristik anzeigt, dass die Anweisung eine skalare Anweisung ist.

3. System gemäß Anspruch 1, wobei die skalare Charakteristik eine Identifikation der Anweisung als eine skalare Anweisung und eine Identifikation eines Quell-Operanden als einen skalaren Operanden umfasst.

4. System gemäß Anspruch 1, wobei der SIMT-Prozessor ferner konfiguriert ist, die Anweisung als eine skalare Anweisung zu identifizieren, wenn ein Opcode, welcher in der Anweisung umfasst ist, ein skalarer Opcode ist.

5. System gemäß Anspruch 1, wobei der SIMT-Prozessor ferner konfiguriert ist, die Anweisung fur für einen ersten Thread in der Thread-Gruppe auszuliefern, wenn alle der Quelloperanden, welche in der Anweisung umfasst sind, skalare Operanden sind.

6. System gemäß Anspruch 1, wobei der SIMT-Prozessor ferner konfiguriert ist, die Anweisung als eine skalare Anweisung basierend auf Operanden zu identifizieren, welche in der Anweisung umfasst sind.

7. System gemäß Anspruch 1, wobei der SIMT-Prozessor ferner konfiguriert ist, einen Quelloperanden, welcher in der Anweisung umfasst ist, als einen skalaren Operanden zu identifizieren, welcher von einem Quelloperanden-Register für alle der Threads in der Thread-Gruppe gelesen ist.

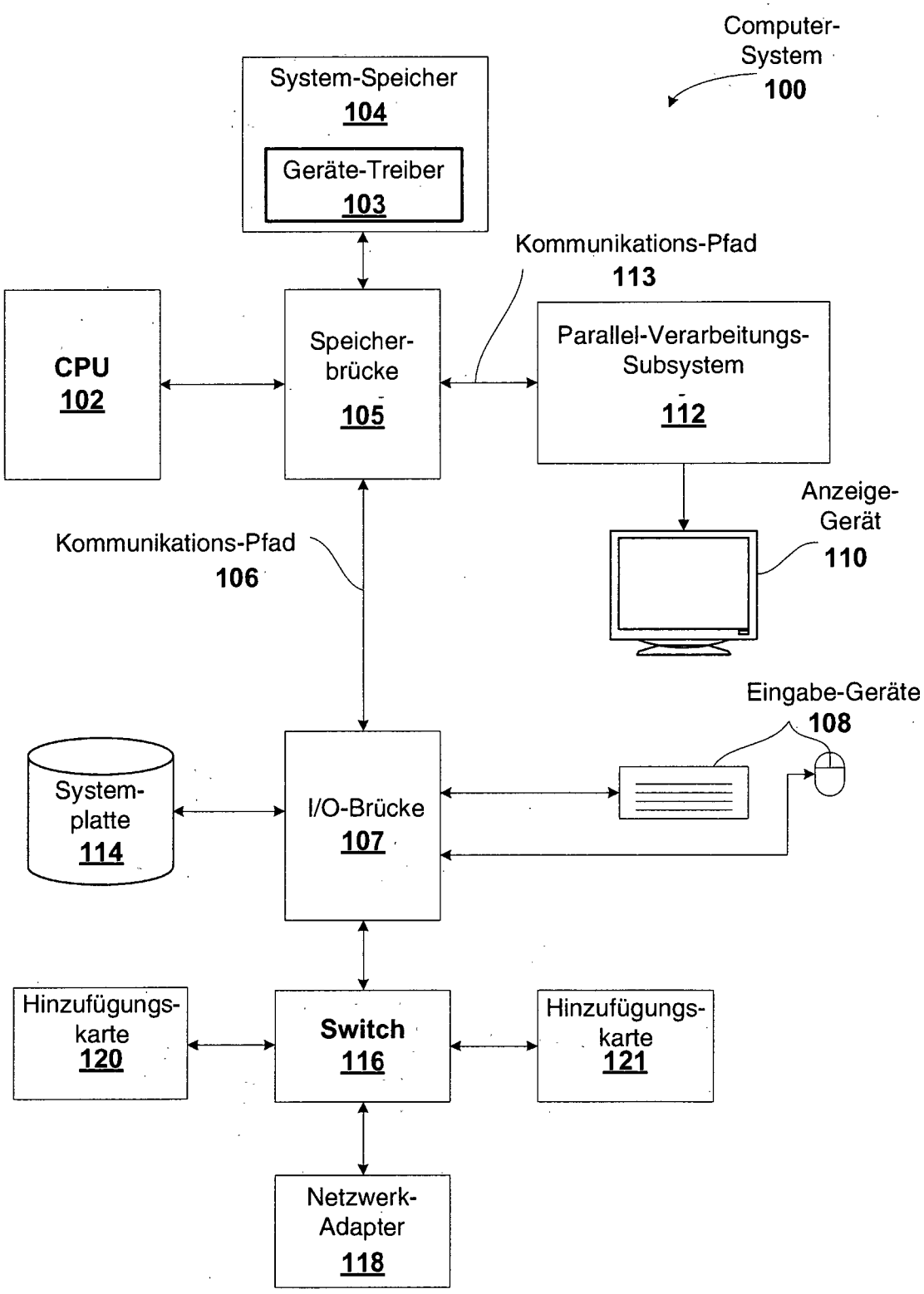
8. System gemäß Anspruch 1, wobei der SIMT-Prozessor ferner konfiguriert ist, einen Quelloperanden, welcher in der Anweisung umfasst ist, von einem Quelloperanden-Register nur für einen ersten Thread in der Thread-Gruppe zu lesen, welcher aktiv ist, wenn die skalare Charakteristik anzeigt, dass der Quelloperand ein skalarer Operand ist.

9. System gemäß Anspruch 1, wobei der SIMT-Prozessor ferner konfiguriert ist, ein Destination-Operanden-Register nur einmalig zu schreiben, wenn die skalare Charakteristik anzeigt, dass der Destination-Operand ein skalarer Operand ist.

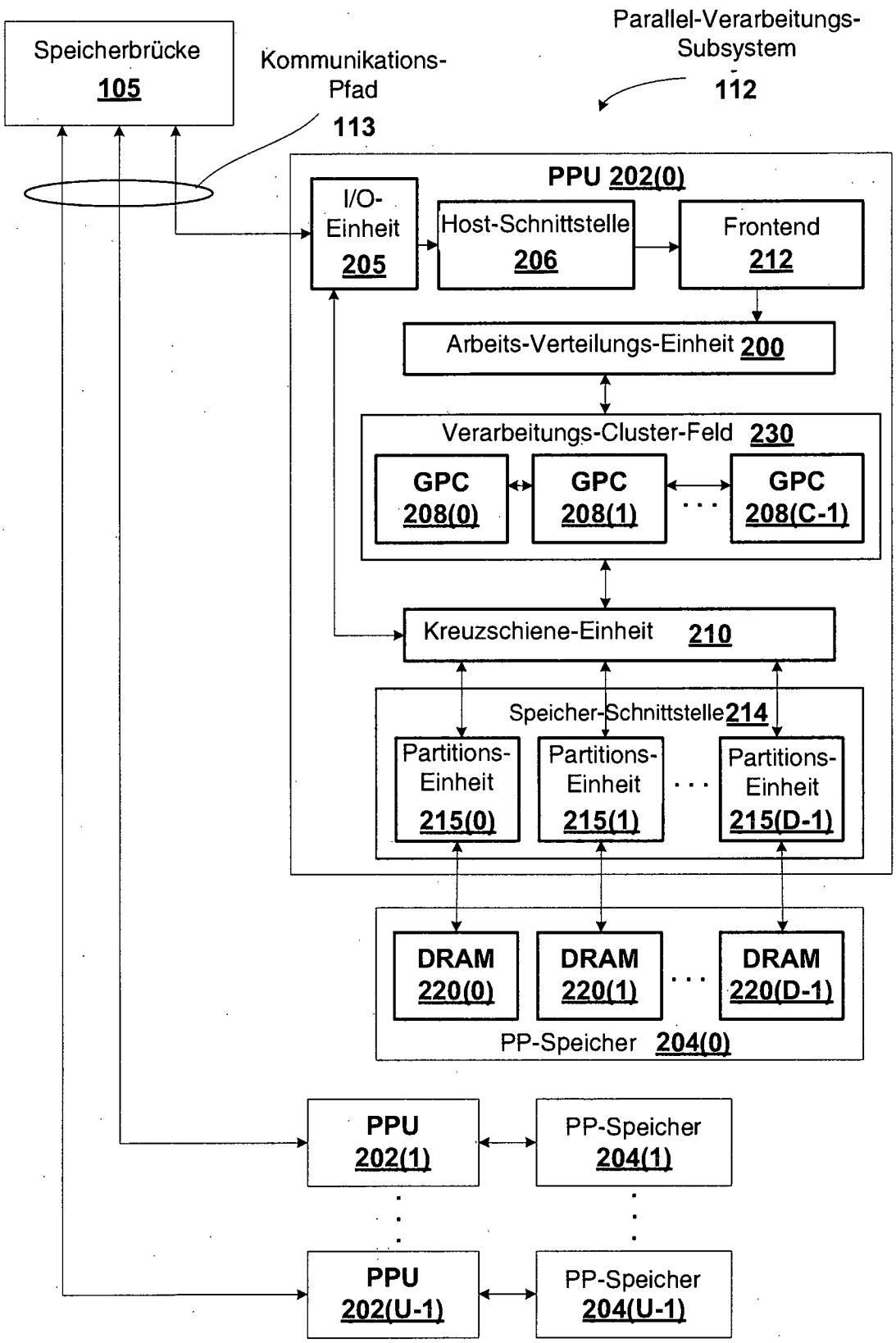
10. System gemäß Anspruch 1, wobei der Teil der Threads in der Thread-Gruppe nur Threads in der Thread-Gruppe umfasst, welche aktiv sind, basierend auf Divergenz-Information.

Es folgen 9 Blatt Zeichnungen

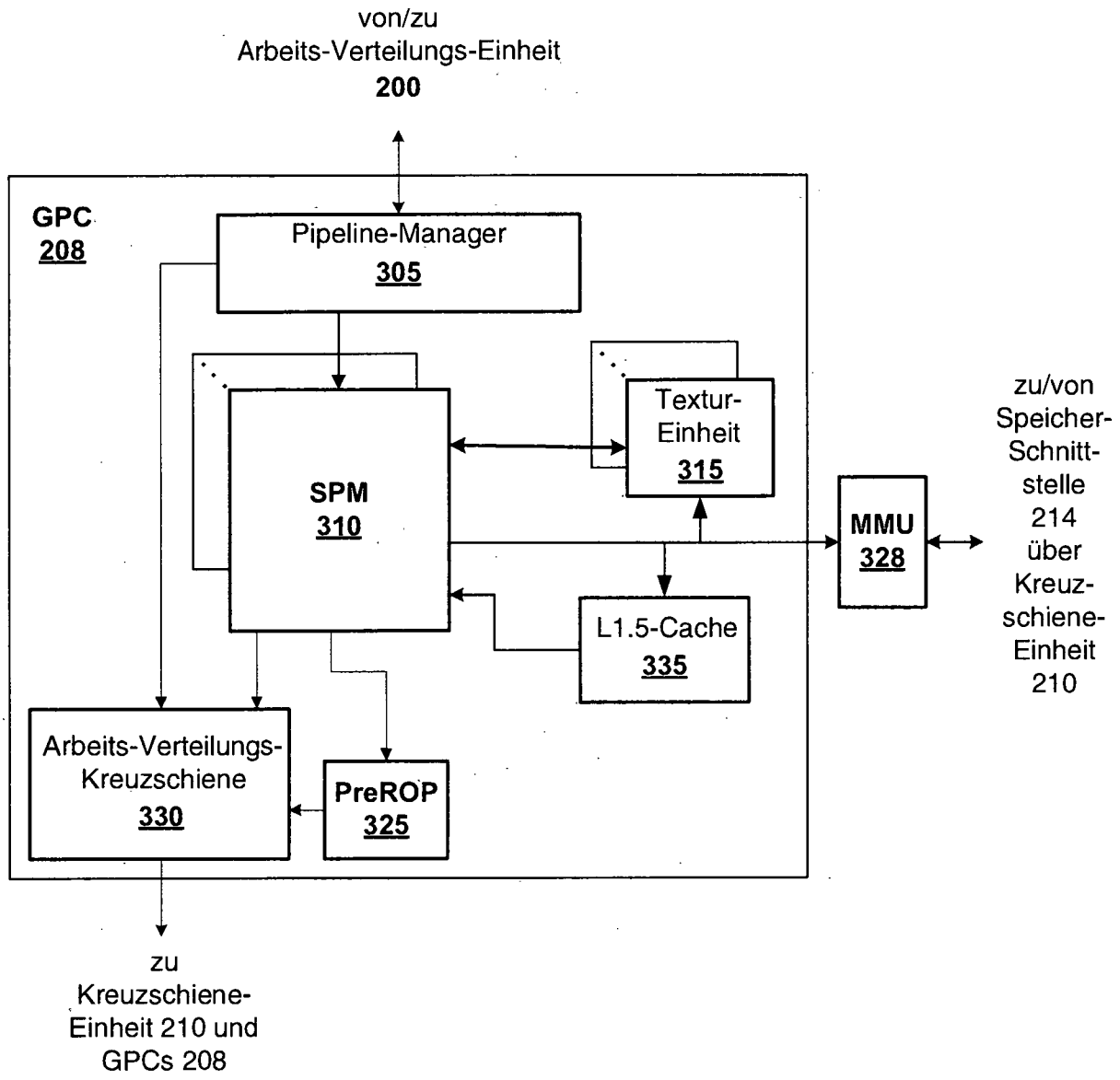
Anhängende Zeichnungen



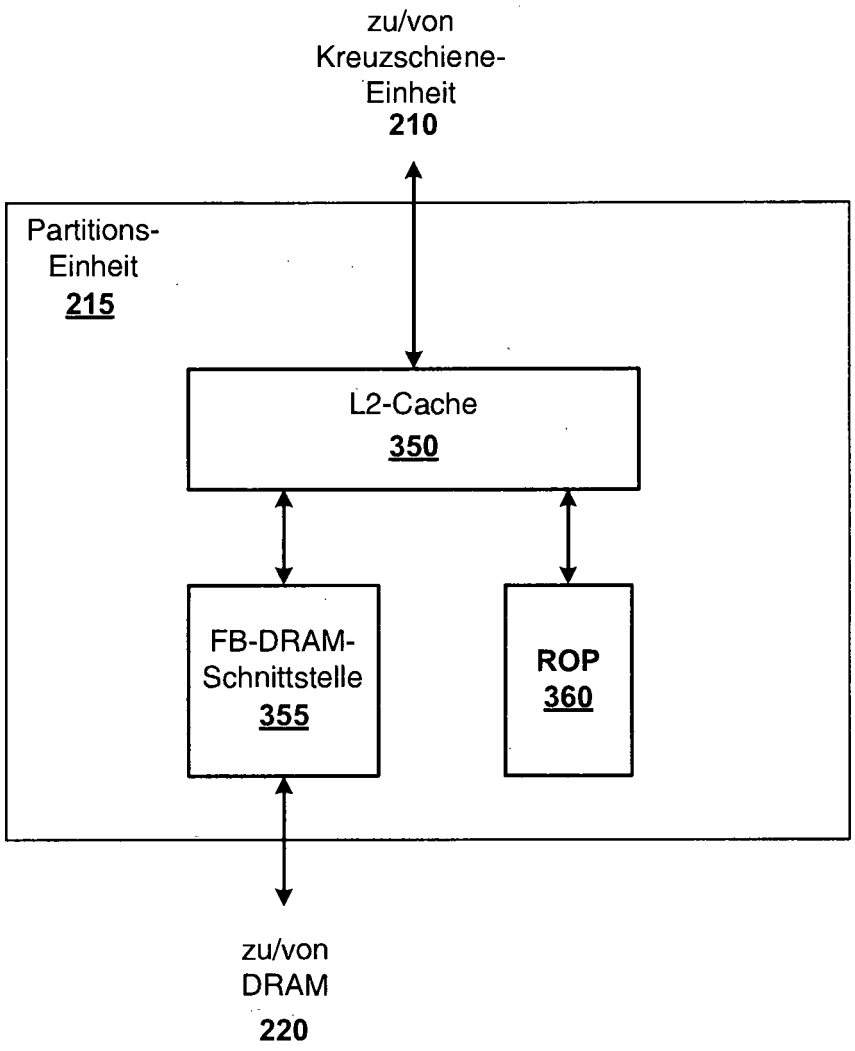
Figur 1


Speicherbrücke
105
Kommunikations-
Pfad
113
Parallel-Verarbeitungs-
Subsystem
112
PPU 202(0)
I/O-
Einheit
205
Host-Schnittstelle
206
Frontend
212
Arbeits-Verteilungs-Einheit 200
Verarbeitungs-Cluster-Feld 230
GPC
208(0)
GPC
208(1)
GPC
208(C-1)
Kreuzschiene-Einheit 210
Speicher-Schnittstelle 214
Partitions-
Einheit
215(0)
Partitions-
Einheit
215(1)
Partitions-
Einheit
215(D-1)
DRAM
220(0)
DRAM
220(1)
DRAM
220(D-1)
PP-Speicher 204(0)
PPU
202(1)
PP-Speicher
204(1)
PPU
202(U-1)
PP-Speicher
204(U-1)

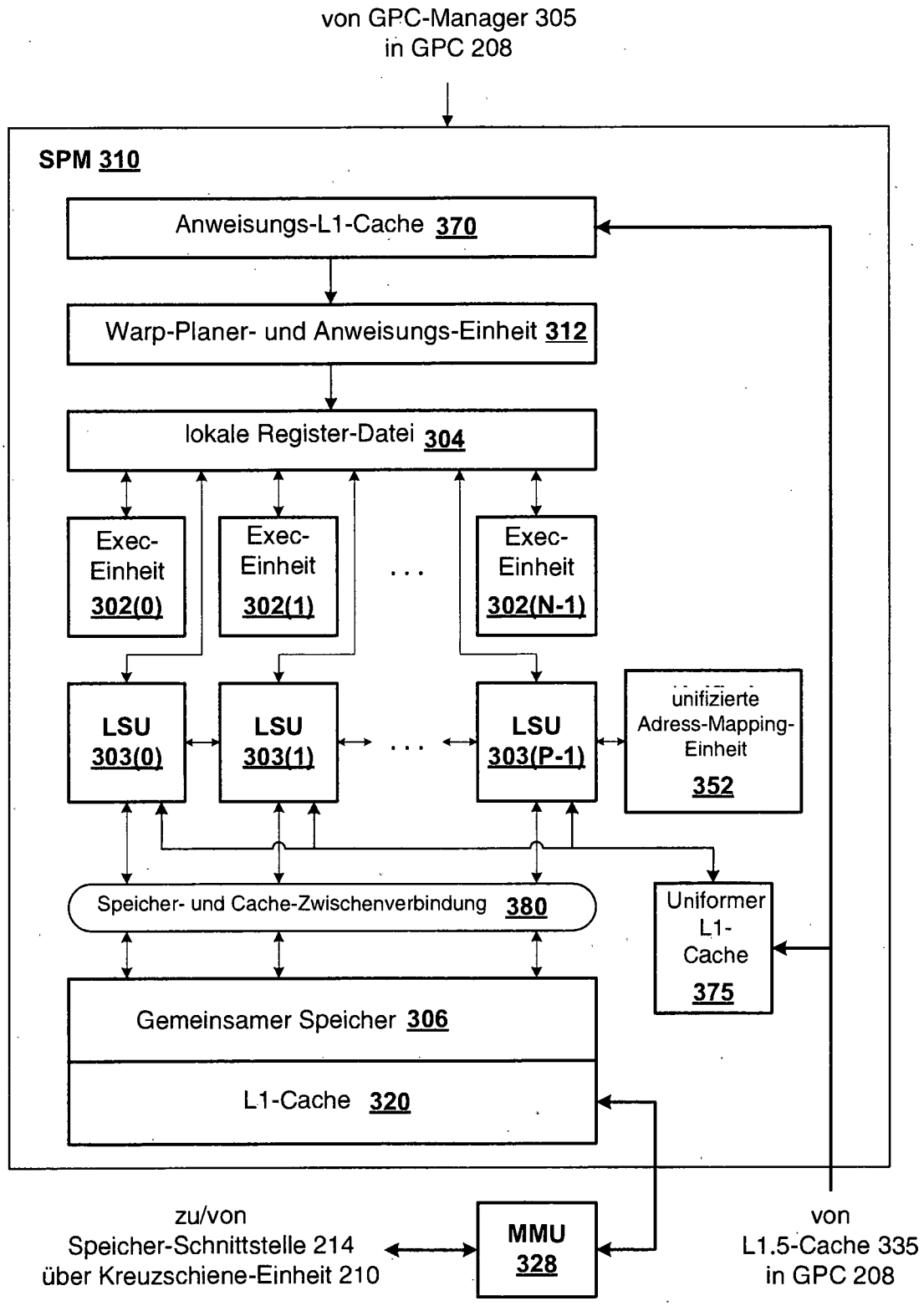
Figur 2


von/zu
Arbeits-Verteilungs-Einheit
200
GPC
208
Pipeline-Manager
305
SPM
310
Textur-
Einheit
315
MMU
328
zu/von
Speicher-
Schnitt-
stelle
214
über
Kreuz-
schiene-
Einheit
210
L1.5-Cache
335
Arbeits-Verteilungs-
Kreuzschiene
330
PreROP
325
zu
Kreuzschiene-
Einheit 210 und
GPCs 208

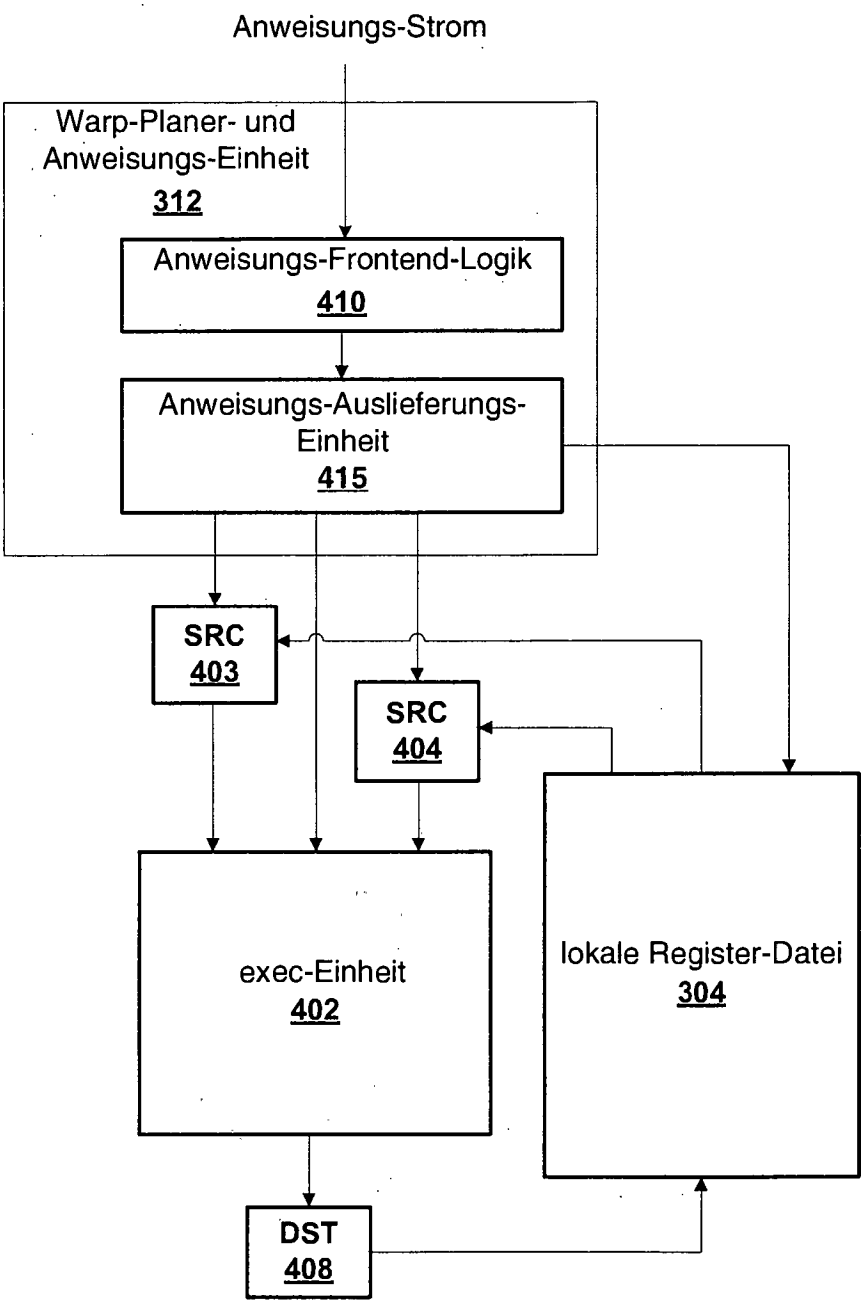
Figur 3A


zu/von
Kreuzschiene-
Einheit
210
Partitions-
Einheit
215
L2-Cache
350
FB-DRAM-
Schnittstelle
355
ROP
360
zu/von
DRAM
220

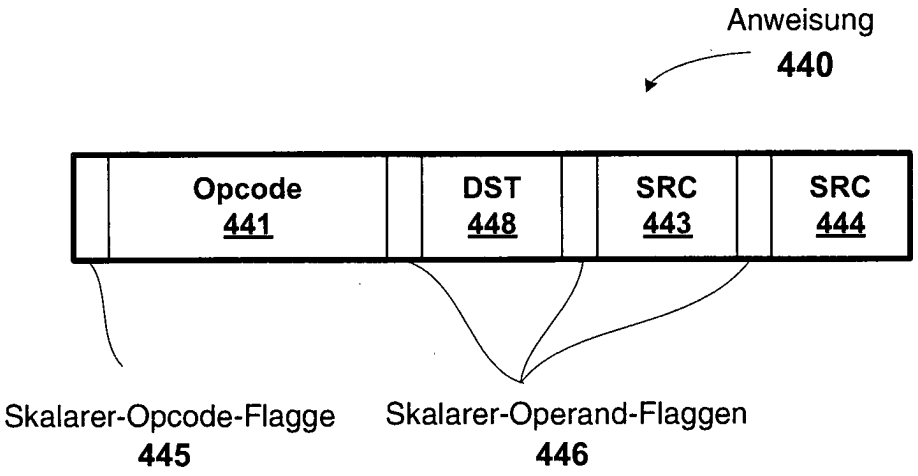
Figur 3B


von GPC-Manager 305
in GPC 208
SPM 310
Anweisungs-L1-Cache 370
Warp-Planer- und Anweisungs-Einheit 312
lokale Register-Datei 304
Exec-Einheit 302(0)
Exec-Einheit 302(1)
Exec-Einheit 302(N-1)
LSU 303(0)
LSU 303(1)
LSU 303(P-1)
unifizierte Adress-Mapping-Einheit 352
Speicher- und Cache-Zwischenverbindung 380
Uniformer L1-Cache 375
Gemeinsamer Speicher 306
L1-Cache 320
MMU 328
zu/von Speicher-Schnittstelle 214 über Kreuzschiene-Einheit 210
von L1.5-Cache 335 in GPC 208

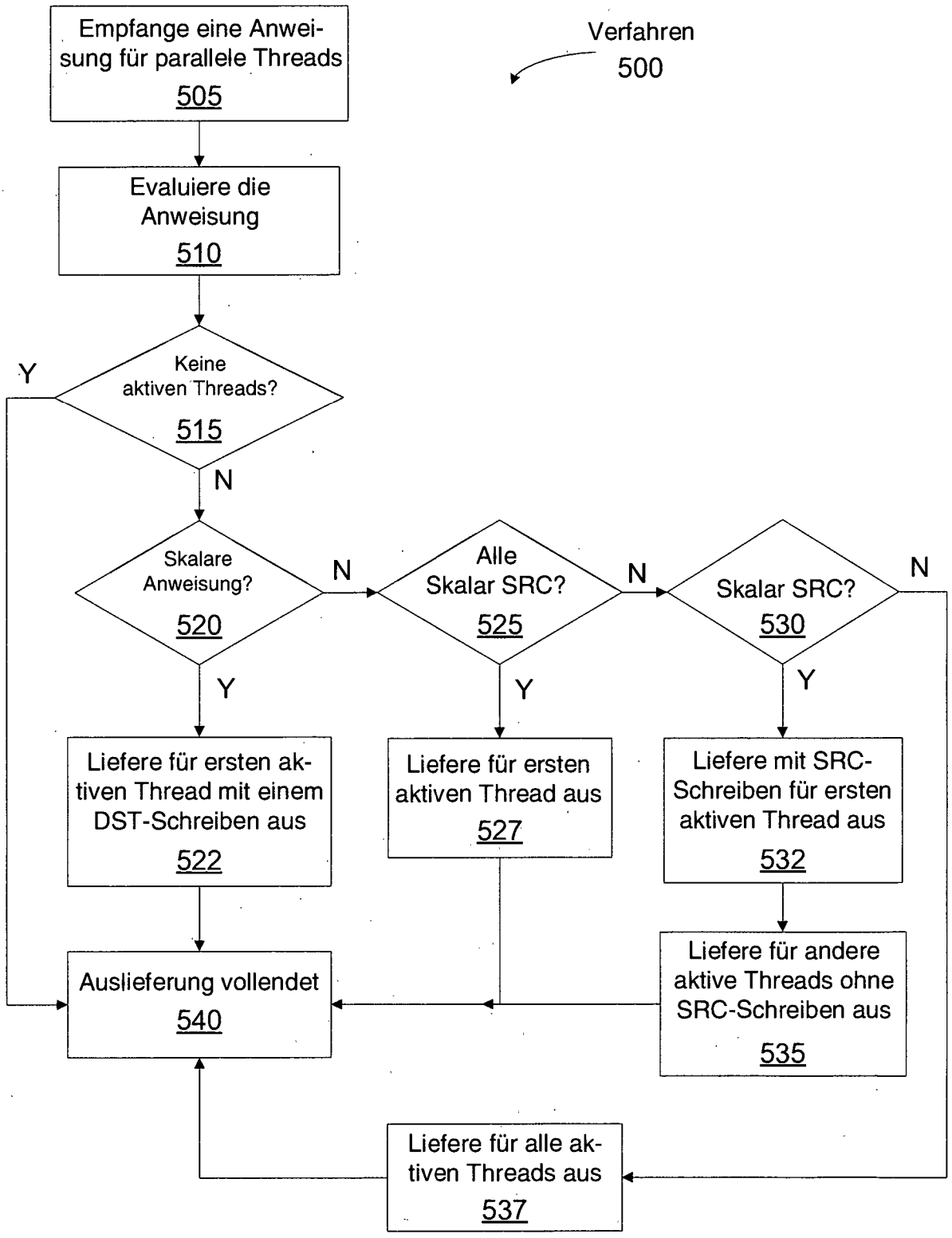
Figur 3C


Anweisungs-Strom
Warp-Planer- und Anweisungs-Einheit
312
Anweisungs-Frontend-Logik
410
Anweisungs-Auslieferungs-Einheit
415
SRC
403
SRC
404
exec-Einheit
402
lokale Register-Datei
304
DST
408

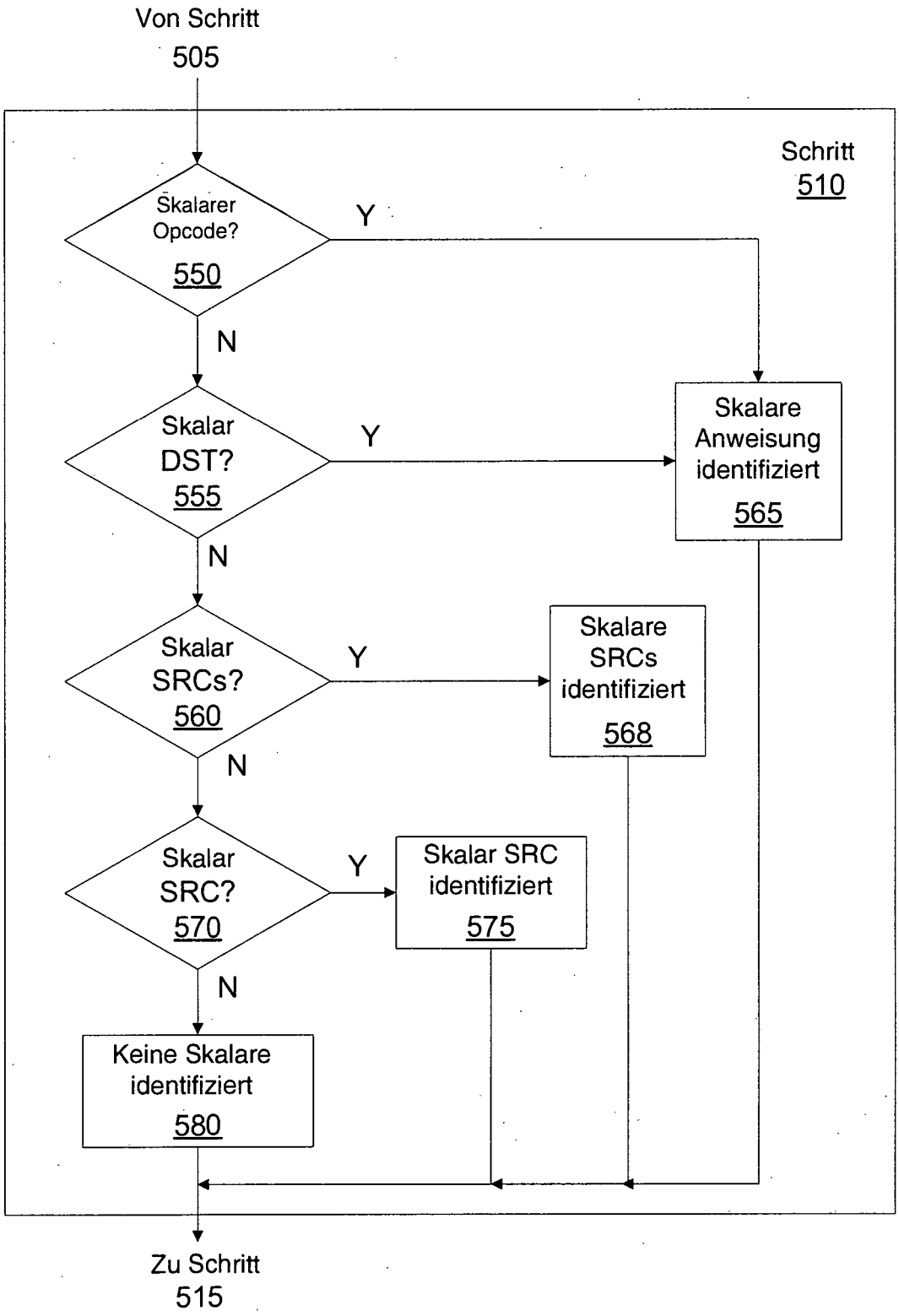
Figur 4A


Anweisung
440
Opcode
441
DST
448
SRC
443
SRC
444
Skalarer-Opcode-Flagge
445
Skalarer-Operand-Flaggen
446

Figur 4B


Empfange eine Anweisung für parallele Threads
505
Verfahren
500
Evaluiere die Anweisung
510
Keine aktiven Threads?
515
Y
N
Skalare Anweisung?
520
N
Alle Skalar SRC?
525
N
Skalar SRC?
530
N
Y
Y
Y
Liefere für ersten aktiven Thread mit einem DST-Schreiben aus
522
Liefere für ersten aktiven Thread aus
527
Liefere mit SRC-Schreiben für ersten aktiven Thread aus
532
Liefere für andere aktive Threads ohne SRC-Schreiben aus
535
Auslieferung vollendet
540
Liefere für alle aktiven Threads aus
537

Figur 5A


Von Schritt
505
Schritt
510
Skalarer Opcode?
550
Y
N
Skalar DST?
555
Y
Skalare Anweisung identifiziert
565
N
Skalar SRCs?
560
Y
Skalare SRCs identifiziert
568
N
Skalar SRC?
570
Y
Skalar SRC identifiziert
575
N
Keine Skalare identifiziert
580
Zu Schritt
515

Figur 5B