(19) Deutsches Patent- und Markenamt

(10) **DE 20 2008 017 916 U1** 2010.12.09

(12) # Gebrauchsmusterschrift

(21) Aktenzeichen: **20 2008 017 916.5**
(22) Anmeldetag: **22.01.2008**
(67) aus Patentanmeldung: **10 2008 005 515.8**
(47) Eintragungstag: **04.11.2010**
(43) Bekanntmachung im Patentblatt: **09.12.2010**

(51) Int Cl.[8]: ***G06F 9/46*** (2006.01)
G06F 9/45 (2006.01)

(30) Unionspriorität:
11/627,892 **26.01.2007** **US**

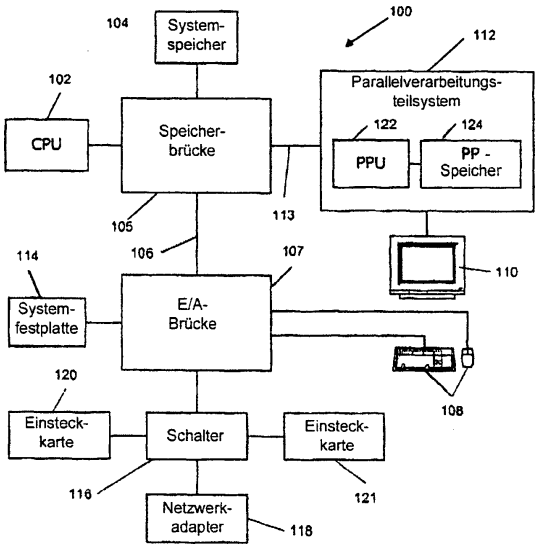
(73) Name und Wohnsitz des Inhabers:
Nvidia Corp., Santa Clara, Calif., US

(74) Name und Wohnsitz des Vertreters:
Dilg Haeusler Schindelmann Patentanwaltsgesellschaft mbH, 80636 München

Die folgenden Angaben sind den vom Anmelder eingereichten Unterlagen entnommen

(54) Bezeichnung: **Virtuelle Architektur und virtueller Befehlssatz für die Berechnung paralleler Befehlsfolgen**

(57) Hauptanspruch: Parallelverarbeitungsarchitektur zum Definieren eines Parallelverarbeitungsvorgangs, wobei die Parallelverarbeitungsarchitektur einen Parallelprozessor und einen Speicher aufweist,
wobei der Speicher einen ersten Programmcode enthält, der eine Abfolge von Operationen definiert, die für jede einer Mehrzahl von virtuellen Befehlsfolgen in einer Gruppierung zusammenwirkender virtueller Befehlsfolgen ausgeführt werden sollen,
wobei der Parallelprozessor betreibbar ist, den ersten Programmcode in ein Programm virtueller Befehlsfolgen zu kompilieren, das eine Abfolge von Befehlen je Befehlsfolge definiert, die für eine repräsentative virtuelle Befehlsfolge der Mehrzahl von virtuellen Befehlsfolgen ausgeführt werden sollen, wobei die Abfolge von Befehlen je Befehlsfolge mindestens einen Befehl enthält, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Mehrzahl von virtuellen Befehlsfolgen definiert; und
wobei der Speicher das Programm virtueller Befehlsfolgen enthält.



Beschreibung

[0001] Die vorliegende Erfindung betrifft allgemein die Parallelverarbeitung und insbesondere eine virtuelle Architektur und einen virtuellen Befehlssatz für die Berechnung paralleler Befehlsfolgen.

[0002] Bei der Parallelverarbeitung arbeiten mehrere Verarbeitungseinheiten (zum Beispiel mehrere Prozessorchips oder mehrere Verarbeitungskerne innerhalb eines einzelnen Chips) gleichzeitig, um Daten zu verarbeiten. Solche Systeme können verwendet werden, um Probleme zu lösen, die sich zur Zerlegung in mehrere Teile anbieten. Ein Beispiel ist die Bildfilterung, wobei jedes Pixel eines ausgegebenen Bildes (oder von ausgegebenen Bildern) aus einer Anzahl von Pixeln eines eingegebenen Bildes (oder von eingegebenen Bildern) berechnet wird. Die Berechnung jedes ausgegebenen Pixels ist allgemein unabhängig von allen anderen, so dass verschiedene Verarbeitungseinheiten verschiedene ausgegebene Pixel parallel berechnen können. Viele andere Arten von Problemen eignen sich ebenfalls für die parallele Zerlegung. Allgemein kann eine parallele N-Wege-Ausführung die Lösung solcher Probleme um ungefähr einen Faktor N beschleunigen.

[0003] Eine weitere Klasse von Problemen eignet sich zur Parallelverarbeitung, wenn die parallelen Ausführungsbefehlsfolgen miteinander koordiniert werden können. Ein Beispiel ist die Schnelle Fouriertransformation (Fast Fourier Transform – FFT), ein rekursiver Algorithmus, bei dem auf jeder Stufe eine Berechnung an den Ergebnissen einer vorherigen Stufe ausgeführt wird, um neue Werte zu generieren, die als Eingaben in die nächste Stufe verwendet werden, bis die Ausgabestufe erreicht ist. Eine einzelne Ausführungsbefehlsfolge kann mehrere Stufen ausführen, solange diese Befehlsfolge verlässlich die Ausgabedaten von vorherigen Stufen erhalten kann. Wenn die Aufgabe zwischen mehreren Befehlsfolgen aufgeteilt werden soll, so muss ein Koordinationsmechanismus vorhanden sein, damit zum Beispiel eine Befehlsfolge nicht versucht, Eingabedaten zu lesen, die noch gar nicht geschrieben wurden. (Eine Lösung dieses Problems ist in der gemeinsam abgetretenen, gleichzeitig anhängigen US-Patentanmeldung Nr. 11/303,780, eingereicht am 15. Dezember 2005, beschrieben).

[0004] Das Programmieren von Parallelverarbeitungssystemen kann jedoch schwierig sein. Der Programmierer muss in der Regel die Anzahl der verfügbaren Verarbeitungseinheiten und ihre Fähigkeiten kennen (Befehlssätze, Anzahl der Datenregister, Zwischenverbindungen usw.), um einen Code zu erzeugen, den die Verarbeitungseinheiten überhaupt ausführen können. Obgleich maschinenspezifische Kompilierer eine große Hilfe auf diesem Gebiet sein können, ist es immer noch erforderlich, den Code jedes Mal neu zu kompilieren, wenn der Code zu einem anderen Prozessor portiert wird.

[0005] Darüber hinaus werden verschiedene Aspekte von Parallelverarbeitungsarchitekturen in rascher Folge hervorgebracht. Zum Beispiel werden ständig neue Plattformarchitekturen, Befehlssätze und Programmiermodelle entwickelt. Wenn sich verschiedene Aspekte der Parallelarchitektur (zum Beispiel das Programmiermodell oder der Befehlssatz) von einer Generation zur nächsten ändern, so müssen auch Anwendungsprogramme, Softwarebibliotheken, Kompilierer und andere Software und Tools entsprechend verändert werden. Diese Instabilität kann einen erheblichen zusätzlichen administrativen Aufwand für die Entwicklung und Pflege von Parallelverarbeitungscode mit sich bringen.

[0006] Wenn eine Koordination zwischen Befehlsfolgen benötigt wird, so wird das parallele Programmieren schwieriger. Der Programmierer muss feststellen, welche Mechanismen in einem bestimmten Prozessor oder Computersystem zur Verfügung stehen, um eine Kommunikation zwischen Befehlsfolgen zu unterstützen (oder zu emulieren), und muss einen Code schreiben, der die verfügbaren Mechanismen ausnutzt. Da die verfügbaren und/oder optimalen Mechanismen auf verschiedenen Computersystemen allgemein verschieden sind, ist ein paralleler Code dieser Art allgemein nicht portierbar. Er muss für jede Hardwareplattform, auf der er läuft, neu geschrieben werden.

[0007] Des Weiteren muss der Programmierer zusätzlich zum Bereitstellen von ausführbarem Code für die Prozessoren noch einen Steuercode für einen ”Master”-Prozessor bereitstellen, der die Abläufe der verschiedenen Verarbeitungseinheiten koordiniert, der zum Beispiel jede Verarbeitungseinheit anweist, welches Programm auszuführen ist und welche Eingabedaten zu verarbeiten sind. Ein solcher Steuercode ist in der Regel für einen bestimmten Master-Prozessor und ein bestimmtes Protokoll für die Kommunikation zwischen Prozessoren spezifisch und muss in der Regel neu geschrieben werden, wenn ein anderer Master-Prozessor verwendet werden soll.

[0008] Die Schwierigkeiten beim Kompilieren und Neukompilieren von Parallelverarbeitungscode können Nutzer davon abschrecken, ihre Systeme entsprechend den Fortschritten der Computertechnologie auf dem modernsten Stand zu halten. Es wäre darum wünschenswert, kompilierten Parallelverarbeitungscode von einer bestimmten Hardwareplattform abzukoppeln und eine stabile Parallelverarbeitungsarchitektur und einen Befehlssatz für interessierende parallele Anwendungen und Tools bereitzustellen.

KURZDARSTELLUNG DER ERFINDUNG

[0009] Ausführungsformen der vorliegenden Erfindung stellen eine virtuelle Architektur und einen virtuellen Befehlssatz für die Berechnung paralleler Befehlsfolgen bereit. Die virtuelle Parallelarchitektur definiert einen virtuellen Prozessor, der die gleichzeitige Ausführung mehrerer virtueller Befehlsfolgen mit mehreren Graden gemeinsamer Datennutzung und Koordination (zum Beispiel Synchronisation) zwischen verschiedenen virtuellen Befehlsfolgen unterstützt, sowie einen virtuellen Ausführungstreiber, der den virtuellen Prozessor steuert. Eine virtuelle Befehlssatzarchitektur für den virtuellen Prozessor wird verwendet, um das Verhalten einer virtuellen Befehlsfolge zu definieren, und enthält Befehle, die sich auf das Verhalten paralleler Befehlsfolgen beziehen, zum Beispiel gemeinsame Datennutzung und Synchronisation. Mit Hilfe der virtuellen parallelen Plattform können Programmierer Anwendungsprogramme entwickeln, in denen virtuelle Befehlsfolgen gleichzeitig ausgeführt werden, um Daten zu verarbeiten. Anwendungsprogramme können in einer hoch-portierbaren Zwischenform gespeichert und verteilt werden, zum Beispiel als Programmcode, der auf die virtuelle parallele Plattform gerichtet ist. Zum Installationszeitpunkt oder Ausführungszeitpunkt passen hardwarespezifische virtuelle Befehlsübersetzer und virtuelle Ausführungstreiber den in einer Zwischenform vorliegenden Anwendungscode an bestimmte Hardware an, auf der er ausgeführt werden soll. Infolge dessen sind Anwendungsprogramme besser portierbar und einfacher zu entwickeln, da der Entwicklungsprozess unabhängig von bestimmter Verarbeitungshardware ist.

[0010] Gemäß einem Aspekt der vorliegenden Erfindung enthält ein Verfahren zum Definieren eines Parallelverarbeitungsvorgangs das Bereitstellen von erstem Programmcode, der eine Abfolge von Operationen definiert, die für jede einer Anzahl virtueller Befehlsfolgen in einer Gruppierung zusammenwirkender virtueller Befehlsfolgen auszuführen sind. Der erste Programmcode wird zu einem Programm virtueller Befehlsfolgen kompiliert, das eine Abfolge von Befehlen je Befehlsfolge definiert, die für eine repräsentative virtuelle Befehlsfolge der Gruppierung auszuführen sind, und die Abfolge von Befehlen je Befehlsfolge enthält mindestens einen Befehl, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Gruppierung definiert. Das Programm virtueller Befehlsfolgen wird gespeichert (zum Beispiel im Speicher oder auf einer Festplatte) und kann anschließend in eine Abfolge von Befehlen übersetzt werde, die einer Zielplattformarchitektur entspricht.

[0011] Außerdem kann noch ein zweiter Programmcode bereitgestellt werden, um eine Gruppierung zusammenwirkender virtueller Befehlsfolgen zu definieren, die dafür geeignet sind, einen Eingabedatensatz zu verarbeiten, um einen Ausgabedatensatz zu erzeugen, wobei jede virtuelle Befehlsfolge in der Gruppierung gleichzeitig das Programm virtueller Befehlsfolgen ausführt. Der zweite Programmcode wird auf vorteilhafte Weise in eine Abfolge von Funktionsaufrufen in einer Bibliothek virtueller Funktionen umgewandelt, wobei die Bibliothek virtuelle Funktionen enthält, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen initialisieren und deren Ausführung veranlassen. Diese Abfolge von Funktionsaufrufen kann ebenfalls gespeichert werden. Das gespeicherte Programm virtueller Befehlsfolgen und die Abfolge von Funktionsaufrufen kann dann in einen Programmcode übersetzt werden, der auf einer Zielplattformarchitektur ausführbar ist, wobei der ausführbare Programmcode eine oder mehrere Plattformbefehlsfolgen definiert, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen ausführen. Der ausführbare Programmcode kann auf einem Computersystem ausgeführt werden, das mit der Zielplattformarchitektur kompatibel ist, wodurch der Ausgabedatensatz erzeugt wird, der in einem Speichermedium gespeichert werden kann (zum Beispiel Computerspeicher, Festplatte oder dergleichen).

[0012] Wie angemerkt, enthält die Abfolge von Befehlen je Befehlsfolge in dem Code des Programms virtueller Befehlsfolgen vorteilhafterweise mindestens einen Befehl, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Gruppierung definiert. Zum Beispiel könnte die Abfolge von Befehlen je Befehlsfolge aufweisen einen Befehl, die Ausführung von Operationen für die repräsentative virtuelle Befehlsfolge an einen bestimmten Punkt in der Abfolge auszusetzen, bis eine oder mehrere der anderen virtuellen Befehlsfolgen jenen bestimmten Punkt erreichen, einen Befehl für die repräsentative virtuelle Befehlsfolge, Daten in einem gemeinsam genutzten Speicher zu speichern, auf den eine oder mehrere der anderen virtuellen Befehlsfolgen Zugriff haben, einen Befehl für die repräsentative virtuelle Befehlsfolge, nicht unterbrechbar (atomically) Daten zu lesen und zu aktualisieren, die in einem gemeinsam genutzten Speicher gespeichert sind, auf den eine oder mehrere der anderen virtuellen Befehlsfolgen Zugriff haben, oder dergleichen.

[0013] Das Programm virtueller Befehlsfolgen kann auch eine Variablendefinitionsaussage enthalten, die eine Variable in einem aus einer Anzahl von virtuellen Zustandsräumen definiert, wobei verschiedene virtuelle Zustandsräume verschiedenen Modi der gemeinsamen Datennutzung zwischen den virtuellen Befehlsfolgen entsprechen. In einer Ausführungsform werden mindestens ein je Befehlsfolge nicht gemeinsam genutzter Modus und ein global gemein-

sam genutzter Modus unterstützt. In anderen Ausführungsformen können auch zusätzliche Modi unterstützt werden, wie zum Beispiel ein gemeinsam genutzter Modus innerhalb einer Gruppierung virtueller Befehlsfolgen und/oder ein gemeinsam genutzter Modus zwischen mehreren Gruppierungen virtueller Befehlsfolgen.

[0014] Gemäß einem weiteren Aspekt der vorliegenden Erfindung enthält ein Verfahren zum Betreiben eines Zielprozessors das Bereitstellen von einem Eingabeprogrammcode. Der Eingabeprogrammcode enthält einen ersten Abschnitt, der eine Abfolge von Operationen definiert, die für jede einer Anzahl virtueller Befehlsfolgen in einer Gruppierung virtueller Befehlsfolgen auszuführen sind, die dafür geeignet sind, einen Eingabedatensatz zu verarbeiten, um einen Ausgabedatensatz zu erzeugen, und enthält auch einen zweiten Abschnitt, der eine Dimension der Gruppierung virtueller Befehlsfolgen definiert. Der erste Abschnitt des Eingabeprogrammcode wird zu einem Programm virtueller Befehlsfolgen kompiliert, das eine Abfolge von Befehlen je Befehlsfolge definiert, die für eine repräsentative virtuelle Befehlsfolge der Gruppierung ausgeführt werden sollen. Die Abfolge von Befehlen je Befehlsfolge enthält mindestens einen Befehl, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Gruppierung definiert. Der zweite Abschnitt des Eingabeprogrammcode wird in eine Abfolge von Funktionsaufrufen an eine Bibliothek virtueller Funktionen umgewandelt, wobei die Bibliothek virtuelle Funktionen enthält, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen initialisieren und deren Ausführung veranlassen. Das Programm virtueller Befehlsfolgen und die Abfolge von Funktionsaufrufen werden in einen Programmcode übersetzt, der auf einer Zielplattformarchitektur ausführbar ist, wobei der ausführbare Programmcode eine oder mehrere reale Befehlsfolgen definiert, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen ausführt. Der ausführbare Programmcode wird auf einem Computersystem ausgeführt, das mit der Zielplattformarchitektur kompatibel ist, wodurch der Ausgabedatensatz erzeugt wird, der in einem Speichermedium gespeichert kann.

[0015] In einigen Ausführungsformen können Gruppierungen virtueller Befehlsfolgen in zwei oder mehr Dimensionen definiert werden. Des Weiteren kann der zweite Abschnitt des Eingabeprogrammcode auch einen Funktionsaufruf enthalten, der eine oder mehrere Dimensionen eines Gitters aus Gruppierungen virtueller Befehlsfolgen definiert, wobei jede Gruppierung in dem Gitter ausgeführt werden soll.

[0016] Es kann jede beliebige Zielplattformarchitektur verwendet werden. In einigen Ausführungsformen enthält die Zielplattformarchitektur einen Master-Prozessor und einen Koprozessor. Während der Übersetzung kann das Programm virtueller Befehlsfolgen in Programmcode übersetzt werden, der parallel durch eine Anzahl von Befehlsfolgen ausführbar ist, die in dem Koprozessor definiert werden, während die Abfolge von Funktionsaufrufen in eine Abfolge von Rufen an ein Treiberprogramm für den Koprozessor, das auf dem Master-Prozessor ausgeführt wird, übersetzt wird. In anderen Ausführungsformen enthält die Zielplattformarchitektur eine zentrale Verarbeitungseinheit (CPU). Während der Übersetzung werden das Programm virtueller Befehlsfolgen und mindestens ein Abschnitt der Abfolge von Funktionsaufrufen in einen Zielprogrammcode übersetzt, der die Gruppierung virtueller Befehlsfolgen mit Hilfe einen Anzahl von CPU-Befehlsfolgen ausführt, die weniger sind als die Anzahl virtueller Befehlsfolgen.

[0017] Gemäß einer weiteren Ausführungsform der vorliegenden Erfindung enthält ein Verfahren zum Betreiben eines Zielprozessors das Erlangen eines Programms virtueller Befehlsfolgen, die eine Abfolge von Befehlen je Befehlsfolge definieren, die für eine repräsentative virtuelle Befehlsfolge einer Anzahl virtueller Befehlsfolgen in einer Gruppierung virtueller Befehlsfolgen ausgeführt werden sollen, die dafür geeignet sind, einen Eingabedatensatz zu verarbeiten, um einen Ausgabedatensatz zu erzeugen. Die Abfolge von Befehlen je Befehlsfolge enthält mindestens einen Befehl, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Gruppierung definiert. Ein zusätzlicher Programmcode, der Dimensionen der Gruppierung virtueller Befehlsfolgen definiert, wird ebenfalls erhalten. Das Programm virtueller Befehlsfolgen und der zusätzliche Programmcode werden in einen Programmcode übersetzt, der auf der Zielplattformarchitektur ausführbar ist, wobei der ausführbare Programmcode eine oder mehrere Plattformbefehlsfolgen definiert, welche die Gruppierung virtueller Befehlsfolgen ausführen. Der ausführbare Programmcode wird auf einem Computersystem ausgeführt, das mit der Zielplattformarchitektur kompatibel ist, wodurch der Ausgabedatensatz erzeugt wird und der Ausgabedatensatz in einem Speicher gespeichert wird.

[0018] In einigen Ausführungsformen kann das Programm virtueller Befehlsfolgen erhalten werden, indem ein Quellprogrammcode empfangen wird, der in einer höheren Programmiersprache geschrieben wurde, und der Quellprogrammcode kompiliert wird, um das Programm virtueller Befehlsfolgen zu generieren. Alternativ kann das Programm virtueller Befehlsfolgen von einem Speichermedium gelesen werden oder von einem räumlich abgesetzten Computersystem über ein Netzwerk empfangen werden. Es versteht sich, dass der Code virtueller Befehlsfolgen,

der gelesen oder empfangen wird, zuvor aus einer höheren Sprache kompiliert worden sein könnte oder direkt als Code generiert worden sein könnte, der mit einer virtuellen Befehlssatzarchitektur kompatibel ist.

[0019] Die folgende detaillierte Beschreibung zusammen mit den begleitenden Zeichnungen ermöglicht ein besseres Verstehen der Art und der Vorteile der vorliegenden Erfindung.

KURZE BESCHREIBUNG DER ZEICHNUNGEN

[0020] Fig. 1 ist ein Blockschaubild eines Computersystems gemäß einer Ausführungsform der vorliegenden Erfindung.

[0021] Fig. 2A und Fig. 2B veranschaulichen die Beziehung zwischen Gittern, Befehlsfolgen-Gruppierungen und Befehlsfolgen in einem Programmiermodell, das in Ausführungsformen der vorliegenden Erfindung verwendet wird.

[0022] Fig. 3 ist ein Blockschaubild einer virtuellen Architektur gemäß einer Ausführungsform der vorliegenden Erfindung.

[0023] Fig. 4 ist ein Konzeptmodell der Verwendung einer virtuellen Architektur zum Betreiben eines Zielprozessors gemäß einer Ausführungsform der vorliegenden Erfindung.

[0024] Fig. 5 ist eine Tabelle, die spezielle Variablen auflistet, die durch eine virtuelle Befehlssatzarchitektur (Instruction Set Architecture – ISA) gemäß einer Ausführungsform der vorliegenden Erfindung definiert wird.

[0025] Fig. 6 ist eine Tabelle, die Typen von Variablen auflistet, die in einer virtuellen ISA gemäß einer Ausführungsform der vorliegenden Erfindung unterstützt werden.

[0026] Fig. 7 ist eine Tabelle, die virtuelle Zustandsräume auflistet, die in einer virtuellen ISA gemäß einer Ausführungsform der vorliegenden Erfindung unterstützt werden.

[0027] Fig. 8A–Fig. 8H sind Tabellen, die virtuelle Befehle auflisten, die in einer virtuellen ISA gemäß einer Ausführungsform der vorliegenden Erfindung definiert werden.

[0028] Fig. 9 ist ein Flussdiagramm eines Prozesses zur Verwendung eines virtuellen Befehlsübersetzers gemäß einer Ausführungsform der vorliegenden Erfindung.

[0029] Fig. 10 ist eine Tabelle, die Funktionen auflistet, die in einer virtuellen Bibliothek für einen virtuellen Ausführungstreiber gemäß einer Ausführungsform der vorliegenden Erfindung verfügbar sind.

DETAILLIERTE BESCHREIBUNG DER ERFINDUNG

[0030] Ausführungsformen der vorliegenden Erfindung stellen eine virtuelle Architektur und einen virtuellen Befehlssatz zur Berechnung paralleler Befehlsfolgen bereit. Die virtuelle Architektur stellt ein Modell eines Prozessors, der die gleichzeitige Ausführung mehrerer Befehlsfolgen mit mehreren Graden gemeinsamer Datennutzung und Koordination (zum Beispiel Synchronisation) zwischen verschiedenen Befehlsfolgen unterstützt, sowie einen virtuellen Ausführungstreiber, der den Modell-Prozessor steuert, bereit. Der virtuelle Befehlssatz, der dafür verwendet wird, das Verhalten einer Verarbeitungsbefehlsfolge zu definieren, enthält Befehle, die sich auf das Verhalten paralleler Befehlsfolgen beziehen, zum Beispiel Befehle, die eine gemeinsame Nutzung von Daten über bestimmte Befehlsfolgen hinweg gestatten, und Befehle, die verlangen, dass unterschiedliche Befehlsfolgen an bestimmten vom Programmierer angegebenen Punkten innerhalb eines Programms synchronisiert werden. Mit Hilfe der virtuellen Plattform können Programmierer Anwendungsprogramme entwickeln, in denen gleichzeitige, zusammenwirkende Befehlsfolgen ausgeführt werden, um Daten zu verarbeiten. Hardware-spezifische virtuelle Befehlsübersetzer und virtuelle Ausführungstreiber passen den Anwendungscode an bestimmte Hardware an, auf der er ausgeführt werden soll. Infolge dessen sind Anwendungsprogramme besser portierbar und einfacher zu entwickeln, da der Entwicklungsprozess unabhängig von bestimmter Verarbeitungshardware ist.

1. Systemüberblick

[0031] Fig. 1 ist ein Blockschaubild eines Computersystems **100** gemäß einer Ausführungsform der vorliegenden Erfindung. Das Computersystem **100** enthält eine zentrale Verarbeitungseinheit (CPU) **102** und einen Systemspeicher **104**, der über einen Buspfad kommuniziert, der eine Speicherbrücke **105** enthält. Die Speicherbrücke **105**, die zum Beispiel ein Northbridge-Chip sein kann, ist über einen Bus oder einen anderen Kommunikationspfad **106** (zum Beispiel einen HyperTransport-Link) mit einer E/A (Eingabe/Ausgabe)-Brücke **107** verbunden. Die E/A-Brücke **107**, die zum Beispiel ein Southbridge-Chip sein kann, empfängt Benutzereingaben von einem oder mehreren Benutzereingabegeräten **108** (zum Beispiel Tastatur, Maus) und leitet die Eingabe über den Pfad **106** und die Speicherbrücke **105** an die CPU **102** weiter. Ein Parallelverarbeitungsteilsystem **112** ist über einen Bus oder einen anderen Kommunikationspfad **113** (zum Beispiel einen PCI Express- oder Accelerated Graphics Port-Link) an die Speicherbrücke **105** gekoppelt. In einer Ausführungsform ist das

Parallelverarbeitungsteilsystem **112** ein Grafik-Teilsystem, das Pixel an ein Anzeigegerät **110** (zum Beispiel einen herkömmlichen Kathodenstrahlröhren- oder Flüssigkristallmonitor) ausgibt. Eine Systemfestplatte **114** ist ebenfalls mit der E/A-Brücke **107** verbunden. Ein Schalter **116** stellt Verbindungen zwischen der E/A-Brücke **107** und anderen Komponenten her, wie zum Beispiel einem Netzwerkadapter **118** und verschiedenen Einsteckkarten **120** und **121**. Es können noch andere (nicht ausdrücklich gezeigte) Komponenten, darunter USB- oder andere Portverbindungen, CD-Laufwerke, DVD-Laufwerke und dergleichen, mit der E/A-Brücke **107** verbunden werden. Kommunikationspfade, welche die verschiedenen Komponenten in Fig. 1 untereinander verbinden, können mit Hilfe beliebiger geeigneter Protokolle, wie zum Beispiel PCI (Peripheral Component Interconnect), PCI Express (PCI-E), AGP (Accelerated Graphics Port), HyperTransport oder sonstiger anderer Bus- oder Punkt-zu-Punkt-Kommunikationsprotokolle implementiert werden, und Verbindungen zwischen verschiedenen Geräten können mit verschiedenen Protokollen arbeiten, wie es dem Fachmann bekannt ist.

[0032] Das Parallelverarbeitungsteilsystem **112** enthält eine Parallelverarbeitungseinheit (Parallel Processing Unit – PPU) **122** und einen Parallelverarbeitungs (Parallel Processing – PP)-Speicher **124**, die zum Beispiel unter Verwendung einer oder mehrerer integrierter Schaltkreiselemente implementiert werden können, wie zum Beispiel programmierbare Prozessoren, anwendungsspezifische integrierte Schaltkreise (Application-Specific Integrated Circuits – ASICs) und Speicherbausteine. Die PPU **122** implementiert vorteilhafterweise einen hoch-parallelen Prozessor, der einen oder mehrere Verarbeitungskerne enthält, von denen jeder in der Lage ist, eine große Anzahl (zum Beispiel Hunderte) von Befehlsfolgen gleichzeitig auszuführen. Die PPU **122** kann dafür programmiert werden, ein weites Feld von Berechnungen auszuführen, zum Beispiel lineare und nicht-lineare Datentransformierungen, Filterung von Video- und/oder Audiodaten, Modellierung (zum Beispiel Anwendung physikalischer Gesetze zum Bestimmen von Position, Geschwindigkeit und anderen Attributen von Objekten), Bildrendern und so weiter. Die PPU **122** kann Daten aus dem Systemspeicher **104** und/oder dem PP-Speicher **124** in einen internen Speicher übertragen, die Daten verarbeiten und Ergebnisdaten zurück in den Systemspeicher **104** und/oder PP-Speicher **124** schreiben, wo andere Systemkomponenten, einschließlich beispielsweise der CPU **102**, auf solche Daten zugreifen können. In einigen Ausführungsformen ist die PPU **122** ein Grafikprozessor, der auch dafür konfiguriert werden kann, verschiedene Aufgaben auszuführen, die im Zusammenhang stehen mit: der Generierung von Pixeldaten aus Grafikdaten, die durch die CPU **102** und/oder den Systemspeicher **104** über die Speicherbrücke **105** und den Bus **113** herangeführt werden; der Interaktion mit dem PP-Speicher **124** (der als Grafikspeicher verwendet werden kann, einschließlich beispielsweise als herkömmlicher Frame-Puffer) zum Speichern und Aktualisieren von Pixeldaten; der Zuführung von Pixeldaten zum Anzeigegerät **110** und dergleichen. In einigen Ausführungsformen kann das PP-Teilsystem **112** eine PPU **122**, die als ein Grafikprozessor fungiert, und eine weitere PPU **122**, die für Allzweckberechnungen verwendet wird, enthalten. Die PPUs können identisch oder verschieden sein, und jede PPU kann ihre eigenen dedizierten PP-Speicherbausteine haben.

[0033] Die CPU **102** fungiert als der Master-Prozessor des Systems **100** und steuert und koordiniert die Operationen anderer Systemkomponenten. Insbesondere gibt die CPU **102** Befehle aus, welche die Funktion der PPU **122** steuern. In einigen Ausführungsformen schreibt die CPU **102** einen Befehlsstrom für die PPU **122** in einen Befehlspuffer, der sich im Systemspeicher **104**, im PP-Speicher **124** oder einem anderen Speicherort befinden kann, auf den sowohl die CPU **102** als auch die PPU **122** zugreifen kann. Die PPU **122** liest den Befehlsstrom aus dem Befehlspuffer und führt Befehle asynchron mit dem Betrieb der CPU **102** aus.

[0034] Es versteht sich, dass das im vorliegenden Text gezeigte System veranschaulichend ist und dass Variationen und Modifikationen möglich sind. Die Verbindungstopologie, einschließlich der Anzahl und Anordnung von Brücken, kann nach Wunsch modifiziert werden. Zum Beispiel ist in einigen Ausführungsformen der Systemspeicher **104** mit der CPU **102** direkt anstatt über eine Brücke verbunden, und andere Geräte kommunizieren mit dem Systemspeicher **104** über die Speicherbrücke **105** und die CPU **102**. In anderen alternativen Topologien ist das PP-Teilsystem **112** mit der E/A-Brücke **107** anstatt mit der Speicherbrücke **105** verbunden. In wieder anderen Ausführungsformen könnten die E/A-Brücke **107** und die Speicherbrücke **105** in einen einzelnen Chip integriert werden. Die im vorliegenden Text gezeigten konkreten Komponenten sind optional. Zum Beispiel könnte eine beliebige Anzahl von Einsteckkarten oder Peripheriegeräten unterstützt werden. In einigen Ausführungsformen wird der Schalter **116** weggelassen, und der Netzwerkadapter **118** und die Einsteckkarten **120**, **121** sind direkt mit der E/A-Brücke **107** verbunden.

[0035] Die Verbindung der PPU **122** mit dem Rest des Systems **100** kann auch variiert werden. In einigen Ausführungsformen ist das PP-System **112** als eine Einsteckkarte implementiert, die in einen Erweiterungsschlitz des Systems **100** eingesteckt werden kann. In anderen Ausführungsformen kann eine PPU auf einem einzelnen Chip mit einer Busbrücke, wie zum Beispiel einer Speicherbrücke **105** oder

E/A-Brücke **107**, integriert sein. In wieder anderen Ausführungsformen können einige oder alle Elemente der PPU **122** in die CPU **102** integriert sein.

[0036] Eine PPU kann mit einer beliebigen Menge lokalem PP-Speicher versehen sein, einschließlich ohne lokalem Speicher, und kann lokalen Speicher und Systemspeicher in jeder beliebigen Kombination verwenden. Zum Beispiel kann die PPU **122** ein Grafikprozessor in einer Ausführungsform mit einer vereinigten Speicherarchitektur (Unified Memory Architecture – UMA) sein. In solchen Ausführungsformen wird wenig oder gar kein dedizierter Grafikspeicher bereitgestellt, und die PPU **122** würde ausschließlich oder fast ausschließlich Systemspeicher verwenden. In UMA-Ausführungsformen kann die PPU in einen Brückenchip integriert sein oder kann als ein diskreter Chip mit einem Hochgeschwindigkeitslink (zum Beispiel PCI-E) vorhanden sein, der die PPU mit dem Brückenchip und dem Systemspeicher verbindet.

[0037] Es versteht sich des Weiteren, dass eine beliebige Anzahl von PPUs in ein System aufgenommen werden kann, zum Beispiel durch Einbinden mehrerer PPUs auf einer einzelnen Einsteckkarte, indem mehrere Einsteckkarten mit dem Pfad **113** verbunden werden und/oder indem eine oder mehrere PPUs direkt mit der Hauptplatine eines System verbunden werden. Mehrere PPUs können parallel betrieben werden, um Daten mit einem höheren Durchsatz zu verarbeiten, als es mit einer einzelnen PPU möglich ist.

[0038] Dem Fachmann ist auch klar, dass eine CPU und eine PPU in einem einzelnen Baustein integriert sein können und dass die CPU und die PPU verschiedene Ressourcen gemeinsam nutzen können, wie zum Beispiel Befehlslogik, Puffer, Cachespeicher, Hauptspeicher, Verarbeitungsmaschinen und so weiter, oder dass separate Ressourcen für die Parallelverarbeitung und andere Operationen bereitgestellt werden können. Dementsprechend könnten beliebige oder alle der Schaltkreise und/oder Funktionen, die im vorliegenden Text als zu der PPU gehörend beschrieben werden, auch in einer in geeigneter Weise ausgestatteten CPU implementiert und durch diese ausgeführt werden.

[0039] Systeme, die PPUs enthalten, können in einer Vielzahl verschiedener Konfigurationen und Formfaktoren implementiert werden, darunter Desktop-, Laptop- oder handgehaltene (handheld) Personalcomputer, Server, Arbeitsplatzrechner, Spielekonsolen, eingebettete Systeme und so weiter.

[0040] Der Fachmann erkennt auch, dass ein Vorteil der vorliegenden Erfindung in einer größeren Unabhängigkeit von bestimmter Computerhardware besteht. Dementsprechend versteht es sich, dass Ausführungsformen der vorliegenden Erfindung mit Hilfe jedes beliebigen Computersystems praktiziert werden können, einschließlich Systemen, die keine PPU enthalten.

2. Überblick – virtuelles Programmiermodell

[0041] In Ausführungsformen der vorliegenden Erfindung ist es wünschenswert, die PPU **122** oder einen oder mehrere andere Prozessoren eines Computersystems einzusetzen, um Allzweckberechnungen unter Verwendung von Befehlsfolgen-Gruppierungen auszuführen. Im Sinne des vorliegenden Textes ist eine ”Befehlsfolge-Gruppierung” eine Gruppe, die aus einer Anzahl (n_0) von Befehlsfolgen besteht, die gleichzeitig dasselbe Programm an einem Eingabedatensatz ausführen, um einen Ausgabedatensatz zu erzeugen. Jeder Befehlsfolge in der Befehlsfolge-Gruppierung ist ein eindeutiger Befehlsfolge-Identifikator (eine ”Befehlsfolge-ID”) zugewiesen, auf den die Befehlsfolge während ihrer Ausführung zugreifen kann. Die Befehlsfolge-ID, die als ein eindimensionaler oder mehrdimensionaler numerischer Wert definiert sein kann (zum Beispiel 0 bis n_0-1), steuert verschiedene Aspekte des Verarbeitungsverhaltens der Befehlsfolge. Zum Beispiel kann eine Befehlsfolge-ID verwendet werden, um zu bestimmen, welcher Abschnitt des Eingabedatensatzes eine Befehlsfolge verarbeiten soll, und/oder um zu bestimmen, welchen Abschnitt eines Ausgabedatensatzes eine Befehlsfolge erzeugen oder schreiben soll.

[0042] In einigen Ausführungsformen sind die Befehlsfolgen-Gruppierungen ”zusammenwirkende” Befehlsfolgen-Gruppierungen oder CTAs (Cooperative Thread Arrays). Wie bei anderen Arten von Befehlsfolgen-Gruppierungen ist eine CTA eine Gruppe mehrerer Befehlsfolgen, die gleichzeitig dasselbe Programm (im vorliegenden Text als ein ”CTA-Programm” bezeichnet) an einem Eingabedatensatz ausführen, um einen Ausgabedatensatz zu erzeugen. In einer CTA können die Befehlsfolgen zusammenwirken, indem sie Daten in einer Weise gemeinsam nutzen, die von der Befehlsfolge-ID abhängt. Zum Beispiel können in einer CTA Daten durch eine Befehlsfolge erzeugt und durch eine andere verbraucht werden. In einigen Ausführungsformen können Synchronisationsbefehle in den CTA-Programmcode an Punkten eingefügt werden, wo Daten gemeinsam genutzt werden sollen, um zu gewährleisten, dass die Daten tatsächlich durch die erzeugende Befehlsfolge erzeugt wurden, bevor die verbrauchende Befehlsfolge versucht, darauf zuzugreifen. Das Ausmaß der gemeinsamen Datennutzung (sofern eine solche stattfindet) zwischen Befehlsfolgen einer CTA wird durch das CTA-Programm bestimmt. Es versteht sich somit, dass in einer bestimmten Anwendung, die CTAs verwendet, die Befehlsfolgen einer CTA je nach dem CTA-Programm Daten gemeinsam nutzen könnten, aber nicht müssen, und die Begriffe

”CTA” und ”Befehlsfolge-Gruppierung” werden im vorliegenden Text synonym verwendet.

[0043] In einigen Ausführungsformen nutzen Befehlsfolgen in einer CTA Eingabedaten und/oder Zwischenergebnisse gemeinsam mit anderen Befehlsfolgen in derselben CTA. Zum Beispiel könnte ein CTA-Programm einen Befehl enthalten, um eine Adresse in einem gemeinsam genutzten Speicher zu berechnen, in den bestimmte Daten geschrieben werden sollen, wobei die Adresse eine Funktion der Befehlsfolge-ID ist. Jede Befehlsfolge berechnet die Funktion unter Verwendung ihrer eigenen Befehlsfolge-ID und schreibt in den entsprechenden Ort. Die Adressfunktion wird vorteilhafterweise so definiert, dass verschiedene Befehlsfolgen in verschiedene Orte schreiben. Solange die Funktion deterministisch ist, ist der Ort, in den eine Befehlsfolge schreibt, vorhersagbar. Das CTA-Programm kann auch einen Befehl enthalten, eine Adresse in dem gemeinsam genutzten Speicher zu berechnen, aus dem Daten gelesen werden sollen, wobei die Adresse eine Funktion der Befehlsfolge-ID ist. Durch Definieren geeigneter Funktionen und Bereitstellen von Synchronisationstechniken können Daten in einer vorhersagbaren Weise durch eine Befehlsfolge einer CTA in einen bestimmten Ort im gemeinsam genutzten Speicher geschrieben werden und durch eine andere Befehlsfolge derselben CTA von diesem Ort gelesen werden. Folglich kann jedes beliebige gewünschte Muster einer gemeinsamen Datennutzung zwischen Befehlsfolgen unterstützt werden, und eine beliebige Befehlsfolge in einer CTA kann Daten mit jeder anderen Befehlsfolge in derselben CTA gemeinsam nutzen.

[0044] CTAs (oder andere Arten von Befehlsfolgen-Gruppierungen) werden vorteilhafterweise verwendet, um Berechnungen auszuführen, die sich für eine datenparallele Zerlegung anbieten. Im Sinne des vorliegenden Textes beinhaltet eine ”datenparallele Zerlegung” jede Situation, bei der ein Rechenproblem durch mehrmaliges paralleles Ausführen desselben Algorithmus an Eingabedaten gelöst wird, um Ausgabedaten zu erzeugen. Zum Beispiel beinhaltet ein häufiger Fall einer datenparallelen Zerlegung das Anwenden desselben Verarbeitungsalgorithmus auf verschiedene Abschnitte eines Eingabedatensatzes, um verschiedene Abschnitte eines Ausgabedatensatzes zu erzeugen. Zu Beispielen von Problemen, die sich für eine datenparallele Zerlegung eignen, gehören Matrixalgebra, lineare und/oder nicht-lineare Transformationen in jeder beliebigen Anzahl von Dimensionen (zum Beispiel Schnelle Fourier-Transformationen) und verschiedenen Filterungsalgorithmen, einschließlich Faltungsfilter in jeder beliebigen Anzahl von Dimensionen, abtrennbare Filter in mehreren Dimensionen und so weiter. Der Verarbeitungsalgorithmus, der auf jeden Abschnitt des Eingabedatensatzes anzuwenden ist, ist in dem CTA-Programm spezifiziert, und jede Befehlsfolge in einer CTA führt dasselbe CTA-Programm an einem einzelnen Abschnitt des Eingabedatensatzes aus. Ein CTA-Programm kann Algorithmen unter Verwendung eines weiten Bereichs mathematischer und logischer Operationen implementieren, und das Programm kann bedingte oder verzweigende Ausführungspfade und direkten und/oder indirekten Speicherzugriff enthalten.

[0045] CTAs und ihre Ausführung sind in weiterer Ausführlichkeit in der oben angesprochenen Anmeldung Nr. 11/303,780 beschrieben.

[0046] In einigen Situationen ist es auch nützlich, ein ”Gitter” aus zueinander in Beziehung stehenden CTAs (oder allgemeiner ausgedrückt: Befehlsfolgen-Gruppierungen) zu definieren. Im Sinne des vorliegenden Textes ist ein ”Gitter” aus CTAs eine Zusammenstellung einer Anzahl (n_1) von CTAs, in der alle CTAs die gleiche Größe (d. h. Anzahl von Befehlsfolgen) haben und dasselbe CTA-Programm ausführen. Die n_1 CTAs innerhalb eines Gitters sind vorteilhafterweise unabhängig voneinander, was bedeutet, dass die Ausführung einer beliebigen CTA in dem Gitter nicht durch die Ausführung einer anderen CTA in dem Gitter beeinflusst wird. Wie noch deutlich werden wird, ermöglicht dieses Merkmal eine signifikante Flexibilität bei der Verteilung von CTAs zwischen verfügbaren Verarbeitungskernen.

[0047] Um verschiedene CTAs innerhalb eines Gitters voneinander zu unterscheiden, wird jeder CTA des Gitters vorteilhafterweise ein ”CTA-Identifikator” (oder eine CTA-ID) zugewiesen. Wie bei Befehlsfolge-IDs kann jeder beliebige eindeutige Identifikator (einschließlich beispielsweise numerischer Identifikatoren) als eine CTA-ID verwendet werden. In einer Ausführungsform sind CTA-IDs einfach sequenzielle (eindimensionale) Indexwerte von 0 bis n_1-1. In anderen Ausführungsformen können mehrdimensionale Indexierungsschemas verwendet werden. Die CTA-ID ist für alle Befehlsfolgen einer CTA gleich, und eine Befehlsfolge einer bestimmten CTA innerhalb des Gitters kann ihre CTA-ID in Verbindung mit ihrer Befehlsfolge-ID verwenden, um zum Beispiel einen Quellenort zum Lesen von Eingabedaten und/oder einen Zielort zum Schreiben von Ausgabedaten zu bestimmen. Auf diese Weise können Befehlsfolgen in verschiedenen CTAs desselben Gitters gleichzeitig am selben Datensatz arbeiten, obgleich in einigen Ausführungsformen die gemeinsame Nutzung von Daten zwischen verschiedenen CTAs in einem Gitter nicht unterstützt wird.

[0048] Das Definieren eines Gitters aus CTAs kann nützlich sein, zum Beispiel wenn es gewünscht wird, mehrere CTAs zu verwenden, um verschiedene Abschnitte eines einzelnen großen Problems zu lösen. Zum Beispiel könnte es wünschenswert sein, einen Filterungsalgorithmus auszuführen, um ein hochauf-

lösendes Fernsehbild (HDTV) zu erzeugen. Wie dem Fachmann bekannt ist, könnte ein HDTV-Bild über 2 Millionen Pixel enthalten. Wenn jede Befehlsfolge ein Pixel erzeugt, so würde die Anzahl der auszuführenden Befehlsfolgen die Anzahl der Befehlsfolgen, die in einer einzelnen CTA verarbeitet werden können, übersteigen (unter der Annahme einer Verarbeitungsplattform mit angemessener Größe und von sinnvollen Kosten, die unter Verwendung herkömmlicher Techniken hergestellt ist).

[0049] Diese große Verarbeitungsaufgabe kann verwaltet werden, indem man das Bild zwischen mehreren CTAs aufteilt, wobei jede CTA einen anderen Abschnitt (zum Beispiel ein 16 × 16-Feld) der ausgegebenen Pixel erzeugt. Alle CTAs führen dasselbe Programm aus, und die Befehlsfolgen verwenden eine Kombination aus der CTA-ID und der Befehlsfolge-ID zum Bestimmen von Orten zum Lesen von Eingabedaten und Schreiben von Ausgabedaten, so dass jede CTA an dem richtigen Abschnitt des Eingabedatensatzes arbeitet und ihren Abschnitt des Ausgabedatensatzes in den richtigen Ort schreibt.

[0050] Es ist anzumerken, dass im Gegensatz zu Befehlsfolgen innerhalb einer CTA (die Daten gemeinsam nutzen können) CTAs innerhalb eines Gitters vorteilhafterweise keine Daten gemeinsam nutzen oder auf sonstige Weise voneinander abhängig sind. Das heißt, zwei CTAs desselben Gitters können sequenziell (in beliebiger Reihenfolge) oder gleichzeitig ausgeführt werden und trotzdem identische Ergebnisse hervorbringen. Folglich kann eine Verarbeitungsplattform (zum Beispiel das System **100** von Fig. 1) ein Gitter aus CTAs ausführen und ein Ergebnis erhalten, indem sie zuerst eine CTA ausführt, dann die nächste CTA, und so weiter, bis alle CTAs des Gitters ausgeführt wurden. Alternativ kann, wenn genügend Ressourcen verfügbar sind, eine Verarbeitungsplattform dasselbe Gitter ausführen und dasselbe Ergebnis erhalten, indem sie mehrere CTAs parallel ausführt.

[0051] In einigen Fällen kann es wünschenswert sein, mehrere (n_2) Gitter aus CTAs zu definieren, wobei jedes Gitter einen anderen Abschnitt eines Datenverarbeitungsprogramms oder einer Datenverarbeitungsaufgabe ausführt. Zum Beispiel könnte die Datenverarbeitungsaufgabe in eine Anzahl von ”Lösungsschritten” unterteilt werden, wobei jeder Lösungsschritt durch Ausführen eines Gitters aus CTAs ausgeführt wird. Als ein weiteres Beispiel könnte die Datenverarbeitungsaufgabe das Ausführen der gleichen oder ähnlichen Operationen an einer Abfolge von Eingabedatensätzen (zum Beispiel aufeinanderfolgenden Frames von Video-Daten) enthalten. Ein Gitter aus CTAs kann für jeden Eingabedatensatz ausgeführt werden. Das virtuelle Programmiermodell unterstützt vorteilhafterweise mindestens diese drei Stufen der Arbeitsdefinition (d. h. Befehlsfolgen, CTAs und Gitter aus CTAs). Gewünschtenfalls könnten auch weitere Stufen unterstützt werden.

[0052] Es versteht sich, dass die Größe (Anzahl n_0 von Befehlsfolgen) einer CTA, die Größe (Anzahl n_1 von CTAs) eines Gitters und die Anzahl (n_2) von Gittern, die zur Lösung eines bestimmten Problems verwendet werden, von den Parametern des Problems und von den Präferenzen des Programmierers oder des automatisierten Agenten, der die Problemzerlegung definiert, abhängen. Somit wird in einigen Ausführungsformen die Größe einer CTA, die Größe eines Gitters und die Anzahl von Gittern vorteilhafterweise durch einen Programmierer definiert.

[0053] Die Probleme, die von dem CTA-Ansatz profitieren, sind in der Regel durch das Vorhandensein einer großen Anzahl von Datenelementen gekennzeichnet, die parallel verarbeitet werden können. In einigen Fällen sind die Datenelemente Ausgabeelemente, von denen jedes durch Ausführen desselben Algorithmus an verschiedenen (eventuell überlappenden) Abschnitten eines Eingabedatensatzes erzeugt wird. In anderen Fällen können die Datenelemente Eingabeelemente sein, die jeweils unter Verwendung desselben Algorithmus zu verarbeiten sind.

[0054] Solche Probleme können immer in mindestens zwei Stufen zerlegt und auf die oben beschriebenen Befehlsfolgen, CTAs und Gitter abgebildet werden. Zum Beispiel könnte jedes Gitter das Ergebnis eines Lösungsschrittes in einer komplexen Datenverarbeitungsaufgabe darstellen. Jedes Gitter ist vorteilhafterweise in eine Anzahl von ”Blöcken” unterteilt, von denen jeder als eine einzelne CTA verarbeitet werden kann. Jeder Block enthält vorteilhafterweise mehrere ”Elemente”, d. h. elementare Abschnitte des zu lösenden Problems (zum Beispiel einen einzelnen Eingabedatenpunkt oder einen einzelnen Ausgabedatenpunkt). Innerhalb der CTA verarbeitet jede Befehlsfolge ein oder mehrere Elemente.

[0055] Die Fig. 2A und Fig. 2B veranschaulichen die Beziehung zwischen Gittern, CTAs und Befehlsfolgen in einem virtuellen Programmiermodell, das in Ausführungsformen der vorliegenden Erfindung verwendet wird. Fig. 2A zeigt eine Anzahl von Gittern **200**, wobei jedes Gitter aus einer zweidimensionalen (2-D) Gruppierung von CTAs **202** hergestellt ist. (Im vorliegenden Text werden mehrere Instanzen gleicher Objekte mit Bezugszahlen bezeichnet, die das Objekt identifizieren, und in Klammern gesetzte Zahlen identifizieren, wo erforderlich, die Instanz.) Wie in Fig. 2B für die CTA **202** (0,0) gezeigt, enthält jede CTA **202** eine 2-D-Gruppierung von Befehlsfolgen (⊖) **204**. Für jede Befehlsfolge **204** in jeder CTA **202** jedes Gitter **200** kann ein eindeutiger Identifikator der Form $I = [i_g, i_c, i_t]$ definiert werden, wobei ein Gitteridentifikator i_g das Gitter eindeutig identifiziert, eine CTA-ID i_c die CTA innerhalb des Gitters eindeutig

identifiziert und eine Befehlsfolge-ID i_t die Befehlsfolge innerhalb der CTA eindeutig identifiziert. In dieser Ausführungsform könnte der Identifikator I aus einem eindimensionalen Gitteridentifikator i_g, einem zweidimensionalen CTA-Identifikator i_c und einem zweidimensionalen Befehlsfolge-Identifikator i_t aufgebaut sein. In anderen Ausführungsformen ist der eindeutige Identifikator I eine Dreiergruppe aus ganzen Zahlen, wobei $0 \leq i_g < n_2$; $0 \leq i_c < n_1$; und $0 \leq i_t < n_0$. In wieder anderen Ausführungsformen könnten beliebige oder alle des Gitters, der CTA und der Befehlsfolge-Identifikatoren als eine eindimensionale ganze Zahl, ein 2D-Koordinatenpaar, eine 3D-Dreiergruppe oder dergleichen ausgedrückt werden. Der eindeutige Befehlsfolge-Identifikator I kann zum Beispiel dafür verwendet werden, einen Quellenort für Eingabedaten innerhalb einer Gruppierung zu bestimmen, die einen Eingabedatensatz für ein ganzes Gitter oder mehrere Gitter umfasst, und/oder um einen Zielort zum Speichern von Ausgabedaten innerhalb einer Gruppierung zu bestimmen, die einen Ausgabedatensatz für ein ganzes Gitter oder mehrere Gitter umfasst.

[0056] Zum Beispiel könnte im Fall eines HDTV-Bildes jede Befehlsfolge **204** einem Pixel des ausgegebenen Bildes entsprechen. Die Größe (Anzahl von Befehlsfolgen **204**) einer CTA **202** kann bei der Problemzerlegung frei entschieden werden und ist nur durch eine Beschränkung auf die Höchstzahl von Befehlsfolgen in einer einzelnen CTA **202** begrenzt (was die endliche Natur der Prozessorressourcen widerspiegelt). Ein Gitter **200** könnte einem ganzen Frame von HDTV-Daten entsprechen, oder mehrere Gitter könnten auf einen einzelnen Frame abgebildet werden.

[0057] In einigen Ausführungsformen ist die Problemzerlegung gleichförmig, was bedeutet, dass alle Gitter **200** die gleiche Anzahl und Anordnung von CTAs **202** haben und alle CTAs **202** die gleiche Anzahl und Anordnung von Befehlsfolgen **204** haben. In anderen Ausführungsformen kann die Zerlegung ungleichförmig sein. Zum Beispiel könnten verschiedene Gitter verschiedene Anzahlen von CTAs enthalten, und verschiedene CTAs (in demselben Gitter oder in verschiedenen Gittern) könnten verschiedene Anzahlen von Befehlsfolgen enthalten.

[0058] Eine CTA, wie oben definiert, kann Dutzende oder sogar Hunderte gleichzeitiger Befehlsfolgen enthalten. Ein Parallelverarbeitungssystem, auf dem eine CTA ausgeführt werden soll, könnte gegebenenfalls eine solche große Anzahl gleichzeitiger Befehlsfolgen unterstützen. In einem Aspekt entkoppelt die vorliegende Erfindung den Programmierer von solchen Hardwarebeschränkungen, indem sie es dem Programmierer gestattet, eine Verarbeitungsaufgabe unter Verwendung des Modells von CTAs und von Gittern aus CTAs unabhängig von den tatsächlichen Fähigkeiten der Hardware zu definieren. Zum Beispiel kann der Programmierer Code (ein "CTA-Programm") schreiben, der die eine oder die mehreren Verarbeitungsaufgaben, die auszuführen sind, durch eine einzelne repräsentative Befehlsfolge der CTA definieren; der eine CTA als eine Anzahl solcher Befehlsfolgen definiert, die jeweils einen eindeutigen Identifikator haben; und der ein Gitter als eine Anzahl von CTAs definiert, die jeweils einen eindeutigen Identifikator haben. Wie unten beschrieben, wird ein solcher Code automatisch in einen Code übersetzt, der auf einer bestimmten Plattform ausgeführt werden kann. Wenn zum Beispiel die CTA so definiert ist, dass sie eine Anzahl n_0 gleichzeitiger Befehlsfolgen enthält, aber die Zielplattform nur eine einzige Befehlsfolge unterstützt, so kann der Übersetzer eine einzelne reale Befehlsfolge definieren, welche die Aufgaben ausführt, die allen der n_0 Befehlsfolgen zugewiesen sind. Wenn die Zielplattform mehr als eine, aber weniger als n_0 gleichzeitige Befehlsfolgen unterstützt, so können die Aufgaben nach Wunsch zwischen der Anzahl verfügbarer Befehlsfolgen aufgeteilt werden.

[0059] Dementsprechend ist das Programmiermodell von CTAs und Gittern als ein virtuelles Modell zu verstehen, d. h. ein Modell, das eine Konzepthilfe für den Programmierer ist und von jeder konkreten physischen Realisierung abgekoppelt ist. Das virtuelle Modell von CTAs und Gittern kann in einer Vielzahl verschiedener Zielplattformen mit variierenden Graden von Hardwareunterstützung für die Parallelverarbeitung realisiert werden. Genauer gesagt, bezieht sich der Begriff "CTA-Befehlsfolge" im Sinne des vorliegenden Textes auf ein virtuelles Modell einer diskreten Verarbeitungsaufgabe (die eventuell mit einer oder mehreren anderen Verarbeitungsaufgaben zusammenwirkt), und es versteht sich, dass CTA-Befehlsfolgen gegebenenfalls eins zu eins auf Befehlsfolgen auf der Zielplattform abgebildet werden könnten.

3. Virtuelle Architektur

[0060] Gemäß einem Aspekt der vorliegenden Erfindung wird eine virtuelle Parallelarchitektur zum Ausführen von CTAs und Gittern aus CTAs definiert. Die virtuelle Parallelarchitektur ist eine Darstellung eines Parallelprozessors und zugehöriger Speicherräume, welche die Ausführung einer großen Anzahl gleichzeitiger CTA-Befehlsfolgen unterstützen, die zu einem Zusammenwirkungsverhalten befähigt sind, wie zum Beispiel der gemeinsamen Nutzung von Daten und der Synchronisierung miteinander an gewünschten Zeitpunkten. Diese virtuelle Parallelarchitektur kann auf eine Vielzahl verschiedener realer Prozessoren und/oder Verarbeitungssysteme abgebildet werden, einschließlich beispielsweise der PPU **122** des Systems **100** von Fig. 1. Die virtuelle Architektur definiert vorteilhafterweise eine Anzahl virtuel-

ler Speicherräume, die verschiedene Grade gemeinsamer Datennutzung und Arten des Zugriffs unterstützen, sowie eine virtuelle Befehlssatzarchitektur (Instruction Set Architecture – ISA), die alle Funktionen identifiziert, die durch einen virtuellen Prozessor ausgeführt werden können. Die virtuelle Architektur definiert vorteilhafterweise auch einen virtuellen Ausführungstreiber, der dafür verwendet werden kann, die CTA-Ausführung zu steuern, zum Beispiel durch Definieren und Starten einer CTA oder eines Gitters aus CTAs.

[0061] Fig. 3 ist ein Blockschaubild einer virtuellen Architektur **300** gemäß einer Ausführungsform der vorliegenden Erfindung. Die virtuelle Architektur **300** enthält einen virtuellen Prozessor **302** mit einem virtuellen Kern **308**, der dafür konfiguriert ist, eine große Anzahl von CTA-Befehlsfolgen parallel auszuführen. Die virtuelle Architektur **300** enthält auch einen globalen Speicher **304**, auf den der virtuelle Prozessor **302** zugreifen kann, und einen virtuellen Treiber **320**, der Befehle ausgibt, um den Betrieb des virtuellen Prozessors **302** zu steuern. Der virtuelle Treiber **320** kann auch auf den globalen Speicher **304** zugreifen.

[0062] Der virtuelle Prozessor **302** enthält ein Front-End **306**, das Befehle von dem virtuellen Treiber **320** empfängt und interpretiert, und einen Ausführungskern **308**, der in der Lage ist, alle n_0 Befehlsfolgen einer einzelnen CTA gleichzeitig auszuführen. Der virtuelle Kern **308** enthält eine große Anzahl (n_0 oder mehr) virtueller Verarbeitungsmaschinen **310**. In einer Ausführungsform führt jede virtuelle Verarbeitungsmaschine **310** eine einzelne CTA-Befehlsfolge aus. Die virtuellen Verarbeitungsmaschinen **310** führen ihre jeweiligen CTA-Befehlsfolgen gleichzeitig aus, wenn auch nicht unbedingt parallel. In einer Ausführungsform spezifiziert die virtuelle Architektur **300** eine Anzahl T (zum Beispiel **384**, **500**, **768** usw.) virtueller Verarbeitungsmaschinen **310**. Diese Anzahl setzt der Anzahl n_0 von Befehlsfolgen in einer CTA eine Obergrenze. Es versteht sich, dass eine Realisierung der virtuellen Architektur **300** auch weniger physische Verarbeitungsmaschinen als die spezifizierte Anzahl T enthalten kann und dass eine einzelne Verarbeitungsmaschine mehrere CTA-Befehlsfolgen ausführen kann, entweder als eine einzelne ”reale” (d. h. Plattform-unterstützte) Befehlsfolge oder als mehrere gleichzeitige reale Befehlsfolgen.

[0063] Der virtuelle Prozessor **302** enthält auch eine virtuelle Befehlseinheit **312**, die dafür sorgt, dass die virtuellen Verarbeitungsmaschinen **310** mit Befehlen für ihre jeweiligen CTA-Befehlsfolgen versorgt werden. Die Befehle werden durch eine virtuelle ISA definiert, die Teil der virtuellen Architektur **300** ist. Ein Beispiel einer virtuellen ISA zur Berechnung paralleler Befehlsfolgen ist unten beschrieben. Die Befehlseinheit **312** verwaltet die Synchronisation der CTA-Befehlsfolgen und andere Zusammenwirkungsaspekte des Verhaltens von CTA-Befehlsfolgen im Verlauf des Sendens von Befehlen an die virtuellen Verarbeitungsmaschinen **310**.

[0064] Der virtuelle Kern **308** stellt eine interne Datenspeicherung mit verschiedenen Zugänglichkeitsgraden bereit. Die speziellen Register **311** können durch die virtuellen Verarbeitungsmaschinen **310** beschrieben, aber nicht gelesen werden, und werden dafür verwendet, Parameter zu speichern, welche die ”Position” jeder CTA-Befehlsfolge innerhalb des Problemzerlegungsmodells von **Fig.** 2 definieren. In einer Ausführungsform enthalten die speziellen Register **311** ein Register je CTA-Befehlsfolge (oder je virtueller Verarbeitungsmaschine **310**), das eine Befehlsfolge-ID speichert. Jedes Befehlsfolge-ID-Register ist nur für eine jeweilige der virtuellen Verarbeitungsmaschinen **310** zugänglich. Die speziellen Register **311** können auch zusätzliche Register enthalten, die durch alle CTA-Befehlsfolgen (oder durch alle virtuellen Verarbeitungsmaschinen **310**) gelesen werden können, die einen CTA-Identifikator, die CTA-Dimensionen, die Dimensionen eines Gitters, zu dem die CTA gehört, und einen Identifikator eines Gitters, zu dem die CTA gehört, speichern. Die speziellen Register **311** werden während der Initialisierung in Reaktion auf Befehle beschrieben, die über das Front-End **306** von dem virtuellen Treiber **320** empfangen werden, und ändern sich während der CTA-Ausführung nicht.

[0065] Lokale virtuelle Register **314** werden durch jede CTA-Befehlsfolge als Arbeitsraum verwendet. Jedes Register wird für die ausschließlich Verwendung einer einzelnen CTA-Befehlsfolge (oder einer einzelnen virtuellen Verarbeitungsmaschine **310**) zugewiesen, und die Daten in jedem der lokalen Register **314** sind nur für die CTA-Befehlsfolge zugänglich, der sie zugewiesen sind. Der gemeinsam genutzte Speicher **316** ist für alle CTA-Befehlsfolgen (innerhalb einer einzelnen CTA) zugänglich. Jeder Ort in dem gemeinsam genutzten Speicher **316** ist für jede CTA-Befehlsfolge innerhalb derselben CTA (oder für jede virtuelle Verarbeitungsmaschine **310** innerhalb des virtuellen Kerns **308**) zugänglich. Der Parameterspeicher **318** speichert Laufzeitparameter (Konstanten), die durch jede CTA-Befehlsfolge (oder jede virtuelle Verarbeitungsmaschine **310**) gelesen, aber nicht beschrieben werden können. In einer Ausführungsform gibt der virtuelle Treiber **320** Parameter an den Parameterspeicher **318** aus, bevor er den virtuellen Prozessor **302** anweist, die Ausführung einer CTA zu beginnen, die diese Parameter verwendet. Jede CTA-Befehlsfolge innerhalb einer CTA (oder einer virtuellen Verarbeitungsmaschine **310** innerhalb des virtuellen Kerns **308**) kann über eine Speicherschnittstelle **322** auf den globalen Speicher **304** zugreifen.

[0066] In der virtuellen Architektur **300** wird der vir-

tuelle Prozessor **302** als ein Koprozessor unter der Steuerung des virtuellen Treibers **320** betrieben. Die Spezifikation der virtuellen Architektur enthält vorteilhafterweise eine virtuelle Anwendungsprogrammschnittstelle (Application Program Interface – API), die Funktionsaufrufe identifiziert, die durch den virtuellen Treiber **320** erkannt werden, und das Verhalten identifiziert, von dem erwartet wird, dass jeder Funktionsaufruf es hervorruft,. Beispielhafte Funktionsaufrufe für eine virtuelle API für die Berechnung paralleler Befehlsfolgen werden unten beschrieben.

[0067] Die virtuelle Architektur **300** kann auf einer Vielzahl verschiedener Hardwareplattformen realisiert werden. In einer Ausführungsform ist die virtuelle Architektur **300** im System **100** von Fig. 1 realisiert, wobei die PPU **122** den virtuellen Prozessor **302** implementiert und ein PPU-Treiberprogramm, das auf der CPU **102** ausgeführt wird, den virtuellen Treiber **320** implementiert. Der globale Speicher **304** kann im Systemspeicher **104** und/oder im PP-Speicher **124** implementiert werden.

[0068] In einer Ausführungsform enthält die PPU **122** einen oder mehrere Verarbeitungskerne, die mit Einzelbefehls-Mehrfachdaten (Single-Instruction, Multiple-Data – SIMD)- und Nebenläufigkeitstechniken arbeiten, um die gleichzeitige Ausführung einer großen Anzahl (zum Beispiel **384** oder **768**) von Befehlsfolgen von einer einzelnen Befehlseinheit (welche die virtuelle Befehlseinheit **312** implementiert) zu unterstützen. Jeder Kern enthält eine Gruppierung P (zum Beispiel 8, 16 usw.) von Parallelverarbeitungsmaschinen **302**, die dafür konfiguriert sind, SIMD-Befehle von der Befehlseinheit zu empfangen und auszuführen, wodurch Gruppen von bis zu P Befehlsfolgen parallel verarbeitet werden können. Der Kern arbeitet mit Nebenläufigkeit (multithreaded), wobei jede Verarbeitungsmaschine in der Lage ist, bis zu einer Anzahl G (zum Beispiel 24) von Befehlsfolgegruppen gleichzeitig auszuführen, zum Beispiel durch Führen von aktuellen Zustandsinformation, die zu jeder Befehlsfolge gehören, dergestalt, dass die Verarbeitungsmaschine rasch von einer Befehlsfolge zu einer anderen umschalten kann. Somit führt der Kern gleichzeitig G SIMD-Gruppen von jeweils P Befehlsfolgen aus, also insgesamt $P \times G$ gleichzeitige Befehlsfolgen. In dieser Realisierung kann es, solange $P \times G \geq n_0$, eine Eins-zu-eins-Entsprechung zwischen den (virtuellen) CTA-Befehlsfolgen und gleichzeitigen Befehlsfolgen geben, die auf der realen PPU **122** ausgeführt werden.

[0069] Spezielle Register **311** können in der PPU **122** implementiert werden, indem man jeden Verarbeitungskern mit einer Registerdatei aus $P \times G$ Einträgen versieht, wobei jeder Eintrag in der Lage ist, eine Befehlsfolge-ID zu speichern, und indem man einen Satz global auslesbarer Register zum Speichern einer CTA-ID, einer Gitter-ID und von CTA- und Gitterdimensionen bereitstellt. Alternativ können die speziellen Register **311** auch unter Verwendung anderer Speicherorte implementiert werden.

[0070] Lokale Register **314** können in der PPU **122** als eine Lokalregisterdatei implementiert werden, die physisch oder logisch in P Bahnen unterteilt ist, die jeweils eine Anzahl von Einträgen aufweisen (wobei jeder Eintrag zum Beispiel ein 32-Bit-Wort speichern könnte). Jeder der P Verarbeitungsmaschinen ist eine Bahn zugewiesen, und entsprechende Einträge in verschiedenen Bahnen können mit Daten für verschiedene Befehlsfolgen, die dasselbe Programm ausführen, gefüllt werden, um die SIMD-Ausführung zu ermöglichen. Verschiedene Abschnitte der Bahnen können verschiedenen der G gleichzeitigen Befehlsfolgegruppen zugewiesen werden, so dass ein bestimmter Eintrag in der Lokalregisterdatei nur für eine bestimmte Befehlsfolge zugänglich ist. In einer Ausführungsform sind bestimmte Einträge innerhalb der Lokalregisterdatei für das Speichern von Befehlsfolge-Identifikatoren reserviert, die eines der speziellen Register **311** implementieren.

[0071] Der gemeinsam genutzte Speicher **316** kann in der PPU **122** als eine gemeinsam genutzte Registerdatei oder als ein gemeinsam genutzter On-Chip-Cachespeicher mit einer Zwischenverbindung implementiert werden, die es jeder Verarbeitungsmaschine gestattet, jeden beliebigen Ort in dem gemeinsam genutzten Speicher zu lesen oder zu beschreiben. Der Parameterspeicher **318** kann in der PPU **122** als eine bezeichnete Sektion innerhalb derselben gemeinsam genutzten Registerdatei oder desselben gemeinsam genutzten Cachespeichers, der den gemeinsam genutzten Speicher **316** implementiert, oder als eine separate gemeinsam genutzte Registerdatei oder ein separater gemeinsam genutzter On-Chip-Cachespeicher implementiert werden, auf den die Verarbeitungsmaschinen einen Nurlesezugriff haben. In einer Ausführungsform wird der Bereich, der den Parameterspeicher implementiert, auch dafür verwendet, die CTA-ID und die Gitter-ID sowie die CTA- und Gitterdimensionen zu speichern, die Abschnitte der speziellen Register **311** implementieren.

[0072] In einer Ausführungsform reagiert ein PPU-Treiberprogramm, das auf der CPU **102** von Fig. 1 ausgeführt wird, auf Funktionsaufrufe der virtuellen API durch Schreiben von Befehlen in einen (nicht ausdrücklich gezeigten) Einspeicherungspuffer im Speicher (zum Beispiel Systemspeicher **104**), aus dem die Befehle durch die PPU **122** ausgelesen werden. Die Befehle sind vorteilhafterweise mit Zustandsparametern verbunden, wie zum Beispiel der Anzahl von Befehlsfolgen in der CTA; dem Ort eines Eingabedatensatzes, der unter Verwendung der CTA zu verarbeiten ist, im globalen Speicher; dem Ort des auszuführenden CTA-Programms im globalen Spei-

cher; und dem Ort im globalen Speicher, in den die Ausgabedaten geschrieben werden sollen. In Reaktion auf die Befehle und Zustandsparameter lädt die PPU **122** Zustandsparameter in einen ihrer Kerne und beginnt dann mit dem Starten von Befehlsfolgen, bis die Anzahl von Befehlsfolgen, die in den CTA-Parametern spezifiziert sind, gestartet wurden. In einer Ausführungsform enthält die PPU **122** eine Steuerlogik, die Befehlsfolge-IDs sequenziell zu Befehlsfolgen in der Reihenfolge zuweist, wie sie gestartet wurden. Die Befehlsfolge-ID kann zum Beispiel an einem bezeichneten Ort innerhalb der Lokalregisterdatei oder in einem speziellen Register, das speziell diesem Zweck dient, gespeichert werden.

[0073] In einer alternativen Ausführungsform ist die virtuelle Architektur **300** in einem einfach-gereihten Verarbeitungskern (zum Beispiel in einigen CPUs) realisiert, der alle CTA-Befehlsfolgen unter Verwendung von weniger als n_0 realen Befehlsfolgen ausführt. Verarbeitungsaufgaben, die das virtuelle Programmiermodell verschiedenen CTA-Befehlsfolgen zuordnet, können zu einer einzelnen Befehlsfolge kombiniert werden, zum Beispiel durch Ausführen der Aufgabe (oder eines Abschnitts der Aufgabe) für eine CTA-Befehlsfolge, dann für die nächste CTA-Befehlsfolge, und so weiter. Vektorausführung, SIMD-Ausführung und/oder sonstige Formen von Parallelismus, die in der Maschine verfügbar sind, können genutzt werden, um Verarbeitungsaufgaben, die mit mehreren CTA-Befehlsfolgen verbunden sind, parallel auszuführen oder um mehrere Verarbeitungsaufgaben, die mit derselben CTA-Befehlsfolge verbunden sind, parallel auszuführen. Somit kann eine CTA unter Verwendung einer einzelnen Befehlsfolge, von n_0 Befehlsfolgen oder einer sonstigen Anzahl von Befehlsfolgen realisiert werden. Wie unten beschrieben, übersetzt ein virtueller Befehlsübersetzer vorteilhafterweise Code, der in die virtuelle Zielarchitektur **300** geschrieben wurde, in Befehle, die für eine Zielplattform spezifisch sind.

[0074] Es versteht sich, dass die im vorliegenden Text beschriebene virtuelle Architektur veranschaulichend ist und dass Variationen und Modifikationen möglich sind. Zum Beispiel kann in einer alternativen Ausführungsform jede virtuelle Verarbeitungsmaschine ein dediziertes Befehlsfolge-ID-Register haben, das die eindeutige Befehlsfolge-ID, die ihrer Befehlsfolge zugewiesen ist, speichert, anstatt Raum in lokalen virtuellen Registern für diesen Zweck zu verwenden.

[0075] Als ein weiteres Beispiel kann die virtuelle Architektur mehr oder weniger Details bezüglich der internen Struktur des virtuellen Kerns **308** spezifizieren. Zum Beispiel könnte spezifiziert werden, dass der virtuelle Kern **308** P nebenläufige virtuelle Verarbeitungsmaschinen enthält, die verwendet werden, um CTA-Befehlsfolgen in P-Wege-SIMD-Gruppen auszuführen, wobei bis zu G SIMD-Gruppen im Kern **308** nebeneinander existieren, dergestalt, dass P × G T bestimmt (die Höchstzahl von Befehlsfolgen in einer CTA). Verschiedene Arten von Speicher und Grade der gemeinsamen Nutzung können ebenfalls spezifiziert werden.

[0076] Die virtuelle Architektur kann in einer Vielzahl verschiedener Computersysteme unter Verwendung einer beliebigen Kombination von Hardware- und/oder Software-Elementen zum Definieren und Steuern jeder Komponente realisiert werden. Obgleich eine Realisierung unter Verwendung von Hardware-Komponenten beispielhaft beschrieben wurde, versteht es sich, dass die vorliegende Erfindung das Abkoppeln von Programmieraufgaben von einer bestimmten Hardware-Realisierung betrifft.

4. Programmieren der virtuellen Architektur

[0077] Fig. 4 ist ein Konzeptmodell **400** der Verwendung der virtuellen Architektur **300** zum Betreiben eines Zielprozessors oder einer Zielplattform **440** gemäß einer Ausführungsform der vorliegenden Erfindung. Wie das Modell **400** zeigt, entkoppelt das Vorhandensein der virtuellen Architektur **300** kompilierte Anwendungen und APIs von der Hardware-Implementierung des Zielprozessors oder der Zielplattform.

[0078] Ein Anwendungsprogramm **402** definiert eine Datenverarbeitungsanwendung, die das oben beschriebene virtuelle Programmiermodell, einschließlich einzelner CTAs und/oder Gitter aus CTAs, nutzt. Allgemein weist das Anwendungsprogramm **402** mehrere Aspekte auf. Als erstes definiert das Programm das Verhalten einer einzelnen CTA-Befehlsfolge. Zweitens definiert das Programm die Dimensionen einer CTA (als Anzahl von CTA-Befehlsfolgen) und, wenn Gitter verwendet werden sollen, die Dimensionen eines Gitters (als Anzahl von CTAs). Drittens definiert das Programm einen Eingabedatensatz, der durch die CTA (oder das Gitter) verarbeitet werden soll, und einen Ort, an dem der Ausgabedatensatz gespeichert werden soll. Viertens definiert das Programm ein Gesamt-Verarbeitungsverhalten, einschließlich beispielsweise, wann jede CTA oder jedes Gitter gestartet werden soll. Das Programm kann zusätzlichen Code enthalten, der dynamisch die Dimensionen einer CTA oder eines Gitters bestimmt; der bestimmt, ob neue CTAs oder Gitter weiter gestartet werden sollen, und so weiter.

[0079] Das Anwendungsprogramm **402** kann in einer höheren Programmiersprache, wie zum Beispiel C/C++, FORTRAN oder dergleichen, geschrieben werden. In einer Ausführungsform spezifiziert ein ”C/C++”-Anwendungsprogramm direkt das Verhalten einer (virtuellen) CTA-Befehlsfolge. In einer weiteren Ausführungsform wird ein Anwendungsprogramm

unter Verwendung einer datenparallelen Sprache geschrieben (zum Beispiel Fortran 90, C* oder Data-Parallel C) und spezifiziert datenparallele Operationen an Gruppierungen und aggregierten Datenstrukturen. Ein solches Programm kann zu virtuellem ISA-Programmcode kompiliert werden, der das Verhalten einer (virtuellen) CTA-Befehlsfolge spezifiziert. Um das Definieren des Verhaltens einer CTA-Befehlsfolge zu ermöglichen, können Spracherweiterungen oder eine Funktionsbibliothek bereitgestellt werden, über die der Programmierer das Verhalten paralleler CTA-Befehlsfolgen spezifizieren kann. Zum Beispiel können spezielle Symbole oder Variablen definiert werden, die der Befehlsfolge-ID, der CTA-ID und der Gitter-ID entsprechen, und es können Funktionen bereitgestellt werden, über die der Programmierer angeben kann, wann die CTA-Befehlsfolge mit anderen CTA-Befehlsfolgen synchronisiert werden sollte.

[0080] Wenn das Anwendungsprogramm **402** kompiliert wird, so erzeugt der Kompilierer **408** einen virtuellen ISA-Code **410** für jene Abschnitte des Anwendungsprogramms **402**, die das Verhalten von CTA-Befehlsfolgen definieren. In einer Ausführungsform wird virtueller ISA-Code **410** in der virtuellen ISA der virtuellen Architektur **300** von Fig. 3 ausgedrückt. Der virtuelle ISA-Code **410** ist Programmcode, wenn auch nicht unbedingt Code in einer Form, der auf einer bestimmten Zielplattform ausgeführt werden kann. Als solches kann der virtuelle ISA-Code **410** wie jeder andere Programmcode gespeichert und/oder verteilt werden. In anderen Ausführungsformen können Anwendungsprogramme ganz oder teilweise als virtueller ISA-Code **410** spezifiziert werden, und der Kompilierer **408** kann ganz oder teilweise umgangen werden.

[0081] Ein virtueller Befehlsübersetzer **412** konvertiert virtuellen ISA-Code **410** in einen Ziel-ISA-Code **414**. In einigen Ausführungsformen ist der Ziel-ISA-Code **414** ein Code, der direkt durch eine Zielplattform **440** ausgeführt werden kann. Zum Beispiel kann, wie durch die in gestrichelter Linie Kästen in Fig. 4 gezeigt, in einer Ausführungsform der Ziel-ISA-Code **414** durch eine Befehlseinheit **430** in der PPU **122** empfangen und korrekt decodiert werden. Je nach den Spezifika der Zielplattform **440** könnte der virtuelle ISA-Code **410** in Code je Befehlsfolge übersetzt werden, um durch jede von n_0 Befehlsfolgen auf der Zielplattform **440** ausgeführt zu werden. Alternativ könnte der virtuelle ISA-Code **410** in einen Programmcode übersetzt werden, um in weniger als n_0 Befehlsfolgen ausgeführt zu werden, wobei jede Befehlsfolge Verarbeitungsaufgaben enthält, die zu mehr als einer der CTA-Befehlsfolgen in Beziehung stehen.

[0082] In einigen Ausführungsformen werden die Definition von Dimensionen von CTAs und/oder Gittern sowie das Definieren von Eingabedatensätzen und Ausgabedatensätzen durch eine virtuelle API gehandhabt. Das Anwendungsprogramm **402** kann Rufe an eine Bibliothek **404** aus virtuellen API-Funktionen enthalten. In einer Ausführungsform wird dem Programmierer eine Spezifikation der virtuellen API (einschließlich beispielsweise Funktionsnamen, Eingaben, Ausgaben und Effekte, aber keine Implementierungsdetails) zur Verfügung gestellt, und der Programmierer arbeitet virtuelle API-Rufe direkt in das Anwendungsprogramm **402** ein, wodurch direkt virtueller API-Code **406** erzeugt wird. In einer weiteren Ausführungsform wird der virtuelle API-Code **406** durch Kompilieren eines Anwendungsprogramms **402** erzeugt, das eine andere Syntax zum Definieren von CTAs und Gittern verwendet.

[0083] Virtueller API-Code **406** wird zum Teil durch Bereitstellen eines virtuellen Ausführungstreibers **416** realisiert, der die virtuellen API-Befehle aus Code **406** in Ziel-API-Befehle **418** übersetzt, die durch die Zielplattform **440** verarbeitet werden können. Zum Beispiel können, wie durch die in Strichlinie dargestellten Kästen in Fig. 4 gezeigt, in einer Ausführungsform die Ziel-API-Befehle **418** durch einen PPU-Treiber **432** empfangen und verarbeitet werden, der entsprechende Befehle an das Front-End **434** der PPU **122** übermittelt. (In dieser Ausführungsform kann der virtuelle Ausführungstreiber **416** ein Aspekt oder Abschnitt des PPU-Treibers **432** sein.) In einer weiteren Ausführungsform braucht der virtuelle Ausführungstreiber nicht einem Treiber für einen Koprozessor zu entsprechen; er könnte einfach ein Steuerprogramm sein, das andere Programme oder Befehlsfolgen auf demselben Prozessor startet, auf dem der virtuelle Ausführungstreiber läuft.

[0084] Es versteht sich, dass ein virtueller Befehlsübersetzer **412** und ein virtueller Ausführungstreiber **416** für jede beliebige Plattform oder Architektur erzeugt werden können, die in der Lage ist, eine CTA-Ausführung zu unterstützen. Insofern virtuelle Befehlsübersetzer **412** für verschiedene Plattformen oder Architekturen aus derselben virtuellen ISA übersetzen können, kann derselbe virtuelle ISA-Code **410** mit jeder beliebigen Plattform oder Architektur verwendet werden. Somit braucht das Anwendungsprogramm **402** nicht für jede mögliche Plattform oder Architektur rekompiliert zu werden.

[0085] Des Weiteren ist es nicht notwendig, dass die Zielplattform **440** eine PPU und/oder einen PPU-Treiber, wie in Fig. 4 gezeigt, enthält. Zum Beispiel ist in einer alternativen Ausführungsform die Zielplattform eine CPU, die Software-Techniken verwendet, um eine gleichzeitige Ausführung einer großen Anzahl von Befehlsfolgen zu emulieren, und der Ziel-ISA-Code und die Ziel-API-Befehle entsprechen Befehlen in einem Programm (oder einer Gruppe von untereinander kommunizierenden Programme), um

durch die Ziel-CPU ausgeführt zu werden, bei der es sich zum Beispiel um eine Einzelkern- oder eine Mehrkern-CPU handeln kann.

5. Beispiel einer virtuellen ISA

[0086] Ein Beispiel einer virtuellen ISA gemäß einer Ausführungsform der vorliegenden Erfindung wird nun beschrieben. Wie oben angemerkt, entspricht die virtuelle ISA vorteilhafterweise dem oben beschriebenen virtuellen Programmiermodell (CTAs und Gittern). Dementsprechend definiert in dieser Ausführungsform der virtuelle ISA-Code **410**, der durch den Kompilierer **408** erzeugt wird, das Verhalten einer einzelnen CTA-Befehlsfolge, um durch eine der virtuellen Verarbeitungsmaschinen **310** im virtuellen Kern **308** von Fig. 3 ausgeführt zu werden. Das Verhalten kann zusammenwirkende Interaktionen mit anderen CTA-Befehlsfolgen enthalten, wie zum Beispiel Synchronisation und/oder gemeinsame Datennutzung.

[0087] Es versteht sich, dass die im vorliegenden Text beschriebene virtuelle ISA allein dem Zweck der Veranschaulichung dient und dass die im vorliegenden Text beschriebenen konkreten Elemente oder Kombinationen von Elementen nicht den Geltungsbereich der Erfindung einschränken. In einigen Ausführungsformen kann ein Programmierer Code in der virtuellen ISA schreiben. In anderen Ausführungsformen schreibt der Programmierer Code in einer anderen höheren Sprache (zum Beispiel FORTRAN, C, C++), und der Kompilierer **408** erzeugt virtuellen ISA-Code. Ein Programmierer kann auch "gemischten" Code schreiben, wobei einige Abschnitte des Codes in einer höheren Sprache und andere Abschnitte in der virtuellen ISA geschrieben sind.

5.1 Spezielle Variablen

[0088] Fig. 5 ist eine Tabelle **500**, die "spezielle" Variablen auflistet, die durch die beispielhafte virtuelle ISA definiert werden (das Präfix "%" wird im vorliegenden Text verwendet, um eine spezielle Variable zu kennzeichnen). Diese Variablen beziehen sich auf das Programmiermodell von **Fig.** 2, wobei jede Befehlsfolge **204** anhand ihrer Position innerhalb einer CTA **202** identifiziert wird, die sich wiederum innerhalb eines bestimmten aus einer Anzahl von Gittern **200** befindet. In einigen Ausführungsformen entsprechen die speziellen Variablen der Tabelle **500** speziellen Registern **311** in der virtuellen Architektur **300** von Fig. 3.

[0089] In Tabelle **500** wird angenommen, dass CTAs und Gitter jeweils in einem dreidimensionalen Raum definiert sind und dass verschiedene Gitter in einem eindimensionalen Raum fortlaufend nummeriert sind. Die virtuelle ISA erwartet, dass die speziellen Variablen von Fig. 5 initialisiert werden, wenn die CTA gestartet wird, und der virtuelle ISA-Code kann einfach diese Variablen ohne Initialisierung verwenden. Die Initialisierung von speziellen Variablen wird unten unter Bezug auf die virtuelle API besprochen.

[0090] Wie in Fig. 5 gezeigt, definiert ein erster 3-Vektor aus speziellen Variablen %ntid = (%ntid.x, %ntid.y, %ntid.z) die Dimensionen (als Anzahl von Befehlsfolgen) einer CTA. Alle Befehlsfolgen einer CTA teilen sich denselben %ntid-Vektor. In der virtuellen Architektur **300** wird erwartet, dass Werte für den %ntid-Vektor an den virtuellen Prozessor **302** über einen Funktionsaufruf einer virtuellen API übermittelt werden, der die Dimensionen einer CTA festlegt, wie unten beschrieben.

[0091] Wie in Fig. 5 gezeigt, bezieht sich ein zweiter 3-Vektor aus speziellen Variablen %tid = (%tid.x, %tid.y, %tid.z) auf die Befehlsfolge-ID einer bestimmten Befehlsfolge innerhalb einer CTA. In der virtuellen Architektur **300** von Fig. 3 wird erwartet, dass der virtuelle Prozessor **302** einen eindeutigen %tid-Vektor zuweist, der die Vorgaben $0 \leq$ %tid.x $<$ %ntid.x, $0 \leq$ %tid.y $<$ %ntid.y und $0 \leq$ %tid.z $<$ %ntid.z erfüllt, wenn jede Befehlsfolge der CTA gestartet wird. In einer Ausführungsform kann der %tid-Vektor so definiert werden, dass er in einem gepackten 32-Bit-Wort gespeichert werden kann (zum Beispiel 16 Bits für %tid.x, 10 Bits für %tid.y und 6 Bits für %tid.z).

[0092] Wie in Fig. 5 gezeigt, definiert ein dritter 3-Vektor aus speziellen Variablen %nctaid = (%nctaid.x, %nctaid.y, %nctaid.z) die Dimensionen (als Anzahl von CTAs) eines Gitters. In der virtuellen Architektur **300** von Fig. 3 wird erwartet, dass die Werte für den %nctaid-Vektor an den virtuellen Prozessor **302** über einen Funktionsaufruf einer virtuellen API übermittelt werden, der die Dimensionen eines Gitters aus CTAs festlegt.

[0093] Wie in Fig. 5 gezeigt, bezieht sich ein vierter 3-Vektor auf spezielle Variablen %ctaid = (%ctaid.x, %ctaid.y, %ctaid.z) auf die CTA-ID einer bestimmten CTA innerhalb eines Gitters. In der virtuellen Architektur **300** von Fig. 3 wird erwartet, dass ein eindeutiger %ctaid-Vektor, der die Vorgaben $0 \leq$ %ctaid.x $<$ %nctaid.x, $0 \leq$ %ctaid.y $<$ %nctaid.y und $0 \leq$ %ctaid.z $<$ %nctaid.z für die CTA erfüllt, an den virtuellen Prozessor **302** übermittelt wird, wenn die CTA gestartet wird.

[0094] Die speziellen Variablen enthalten auch eine skalare %gridid-Variable, die einen Gitteridentifikator für das Gitter bildet, zu dem eine CTA gehört. In der virtuellen Architektur **300** von Fig. 3 wird erwartet, dass ein %gridid-Wert an den virtuellen Prozessor **302** übermittelt wird, um das Gitter zu identifizieren, von dem die momentane CTA ein Teil ist. Der %gridid-Wert wird vorteilhafterweise in virtuellem ISA-Code verwendet, zum Beispiel wenn mehrere Gitter ver-

wendet werden, um verschiedene Abschnitte eines großen Problems zu lösen.

5.2. Programmdefinierte Variablen und virtuelle Zustandsräume

[0095] Die virtuelle ISA ermöglicht es dem Programmierer (oder Kompilierer), eine willkürliche Anzahl von Variablen zu definieren, um in Verarbeitung befindliche Datenelemente darzustellen. Eine Variable wird durch einen Typ und einen "virtuellen Zustandsraum" definiert, der anzeigt, wie die Variable verwendet wird und in welchem Umfang sie gemeinsam genutzt wird. Variablen werden unter Verwendung von Registern oder anderen Speicherstrukturen realisiert, die auf einer Zielplattform verfügbar sind. Auf vielen Zielplattformen kann der Zustandsraum die Wahl der Speicherstruktur beeinflussen, die zum Realisieren einer bestimmten Variable verwendet werden soll.

[0096] Fig. 6 ist eine Tabelle **600**, welche die Variablentypen auflistet, die in der beispielhaften virtuellen ISA-Ausführungsform unterstützt werden. Es werden vier Typen unterstützt: nicht-typisierte Bits, signierte ganze Zahl, unsignierte ganze Zahl und Gleitkomma. Nicht-typisierte Variablen sind einfach einzelne Bits oder Gruppen von Bits der spezifizierten Länge. Signierte und unsignierte ganzzahlige Formate sowie Gleitkommaformate können gemäß herkömmlichen Formaten (zum Beispiel IEEE 754-Standards) definiert werden.

[0097] In dieser Ausführungsform werden mehrere Breiten für jeden Typ unterstützt, wobei der Parameter <n> verwendet wird, um die Breite zu spezifizieren. So zeigt zum Beispiel .s16 eine signierte ganze 16-Bit-Zahl an; .f32 zeigt eine 32-Bit-Gleitkommazahl an; und so weiter. Wie in Tabelle **600** gezeigt, sind einige Variablentypen auf bestimmte Breiten beschränkt. Zum Beispiel müssen Gleitkomma-Variablen mindestens 16 Bits haben, und ganzzahlige Typen müssen mindestens 8 Bits haben. Es wird erwartet, dass eine Realisierung der virtuellen ISA alle spezifizierten Breiten unterstützt. Wenn die Datenpfade und/oder Register des Prozessors schmaler sind als die größte Breite, so können mehrere Register und Prozessorzyklen verwendet werden, um die breiteren Typen zu handhaben, wie dem Fachmann bekannt ist.

[0098] Es versteht sich, dass die im vorliegenden Text verwendeten Datentypen und Breiten veranschaulichend sind und die Erfindung nicht einschränken.

[0099] Fig. 7 ist eine Tabelle, welche die virtuellen Zustandräume auflistet, die in der beispielhaften virtuellen ISA unterstützt werden. Es werden neun Zustandsräume definiert, die verschiedenen Graden der gemeinsamen Nutzung und möglichen Speicherorten in der virtuellen Architektur **300** von Fig. 3 entsprechen.

[0100] Die ersten drei Zustandsräume werden auf der Befehlsfolge-Ebene gemeinsam genutzt, was bedeutet, dass jede CTA-Befehlsfolge eine separate Instanz der Variable hat und keine CTA-Befehlsfolge Zugriff auf die Instanz einer anderen CTA-Befehlsfolge hat. Der Zustandsraum des virtuellen Registers (.reg) wird vorteilhafterweise verwendet, um Operanden, temporäre Werte und/oder Ergebnisse von Berechnungen, die durch jede CTA-Befehlsfolge auszuführen sind, zu definieren. Ein Programm kann jede beliebige Anzahl virtueller Register deklarieren. Virtuelle Register können nur durch einen statischen Kompilierzeitnamen und nicht durch eine berechnete Adresse adressiert werden. Dieser Zustandsraum entspricht lokalen virtuellen Registern **314** in der virtuellen Architektur **300** von Fig. 3.

[0101] Der Zustandsraum des speziellen Registers (.sreg) entspricht den vorgegebenen speziellen Variablen von Fig. 5, die in speziellen Registern **311** in der virtuellen Architektur **300** gespeichert werden. In einigen Ausführungsformen braucht der virtuelle ISA-Code keine anderen Variablen in dem .sreg-Raum zu deklarieren, sondern kann die speziellen Variablen als Eingaben in Berechnungen verwenden. Alle CTA-Befehlsfolgen können alle Variablen in dem .sreg-Zustandsraum lesen. Für %tid (oder seine Komponenten) liest jede CTA-Befehlsfolge ihren eindeutigen Befehlsfolge-Identifikator. Für die anderen Variablen in dem .sreg-Zustandsraum lesen alle CTA-Befehlsfolgen in derselben CTA dieselben Werte.

[0102] Variablen von lokalem Speicher je Befehlsfolge (.local) entsprechen einer Region von globalem Speicher **304**, der für jede CTA-Befehlsfolge einzeln zugewiesen und adressiert wird. Oder anders ausgedrückt: Wenn eine CTA-Befehlsfolge auf eine .local-Variable zugreift, so greift sie auf ihre eigene Instanz der Variable zu, und Änderungen zu einer .local-Variable, die in einer CTA-Befehlsfolge vorgenommen werden, beeinflussen keine anderen CTA-Befehlsfolgen. Im Gegensatz zu den .reg- und .sreg-Zustandsräumen kann lokaler Speicher je Befehlsfolge unter Verwendung berechneter Adressen adressiert werden.

[0103] Die nächsten zwei Zustandsräume definieren Variablen je CTA, was bedeutet, dass jede CTA eine einzelne Instanz der Variable hat, auf die jede ihrer (virtuellen) Befehlsfolgen zugreifen kann. Gemeinsam genutzte (.shared) Variablen können durch jede der CTA-Befehlsfolgen gelesen oder geschrieben werden. In einigen Ausführungsformen wird dieser Zustandsraum auf virtuellen gemeinsam genutzten Speicher **316** der virtuellen Architektur **300** (Fig. 3) abgebildet. In einer Realisierung der virtuel-

len Architektur **300** könnte der .shared-Zustandsraum auf eine Implementierung eines auf dem Chip befindlichen, gemeinsam genutzten Speichers (zum Beispiel eine gemeinsam genutzte Registerdatei oder einen gemeinsam genutzten Cachespeicher) abgebildet werden, während in anderen Realisierungen der .shared-Zustandsraum auf eine Region je CTA von Off-Chip-Speicher abgebildet werden könnte, die wie jeder andere global zugängliche Speicher zugewiesen und adressiert wird.

[0104] Parameter (.param)-Variablen können nur gelesen werden und können durch jede beliebige (virtuelle) Befehlsfolge in der CTA gelesen werden. Dieser Zustandsraum bildet den Parameterspeicher **318** der virtuellen Architektur **300** und kann zum Beispiel in einem auf dem Chip angeordneten gemeinsam genutzten Parameterspeicher oder Cachespeicher oder in einer Region auf global zugänglichem Off-Chip-Speicher realisiert werden, der wie jeder andere global zugängliche Speicher zugewiesen und adressiert wird. Es wird erwartet, dass diese Variablen in Reaktion auf Treiberbefehle vom virtuellen Treiber **320** initialisiert werden.

[0105] Der Konstanten (.const)-Zustandsraum wird zum Definieren von Konstanten je Gitter verwendet, die durch jede beliebige (virtuelle) Befehlsfolge in jeder beliebigen CTA in dem Gitter gelesen (aber nicht modifiziert) werden können. In der virtuellen Architektur **300** kann der .const-Zustandsraum auf eine Region im globalen Speicher abgebildet werden, auf die die CTA-Befehlsfolgen einen Nurlesezugriff haben. Der .const-Zustandsraum kann in einem auf dem Chip befindlichen gemeinsam genutzten Parameterspeicher oder Cachespeicher oder in einer Region je Gitter von global zugänglichem Off-Chip-Speicher realisiert werden, die wie jeder andere global zugängliche Speicher zugewiesen und adressiert wird. Wie beim .param-Zustandsraum wird erwartet, dass Variablen in dem .const-Zustandsraum in Reaktion auf Treiberbefehle vom virtuellen Treiber **320** initialisiert werden.

[0106] Die übrigen drei Zustandsräume definieren ”Kontext”-Variablen, die für jede (virtuelle) Befehlsfolge in jeder CTA, die zu der Anwendung gehört, zugänglich sind. Diese Zustandsräume werden auf einem globalen Speicher **304** in der virtuellen Architektur **300** abgebildet. Globale (.global) Variablen können für allgemeine Zwecke verwendet werden. In einigen Ausführungsformen können auch spezifische Zustandsräume für gemeinsam genutzte Texturen (.tex) und Oberflächen (.surf) definiert werden. Diese Zustandsräume, die zum Beispiel für Grafik-bezogene Anwendungen nützlich sein können, können dafür verwendet werden, Zugang zu Grafiktextur- und Pixeloberflächendatenstrukturen zu definieren und zu ermöglichen, die Datenwerte bereitstellen, die jedem Pixel einer 2-D-(oder in einigen Ausführungsformen einer 3-D-)Gruppierung entsprechen.

[0107] In dem virtuellen ISA-Code **410** von Fig. 4 werden Variablen deklariert, indem der Zustandsraum, der Typ und ein Name spezifiziert werden. Der Name ist ein Platzhalter und durch den Programmierer oder Kompilierer ausgewählt werden. So deklariert zum Beispiel:
.reg .b32 vrl
eine nicht-typisierte Variable von 32 Bits in dem Zustandsraum des virtuellen Registers mit der Bezeichnung vrl. Nachfolgende Zeilen aus virtuellem ISA-Code können sich auf vrl zum Beispiel als eine Quelle oder einen Zielort für eine Operation beziehen.

[0108] Die beispielhafte virtuelle ISA unterstützt auch Gruppierungen und Vektoren virtueller Variablen. Zum Beispiel deklariert
.global .f32 resultArray[1000][1000]
eine virtuelle, global zugängliche 1000-mal-1000-Gruppierung aus 32-Bit-Gleitkommazahlen. Der virtuelle Befehlsübersetzer **412** kann Gruppierungen in adressierbare Speicherregionen abbilden, die dem zugewiesen Zustandsraum entsprechen.

[0109] Vektoren können in einer Ausführungsform unter Verwendung eines Vektor-Prefix .v<w> definiert werden, wobei m die Anzahl der Komponenten des Vektors ist. Zum Beispiel deklariert:
.reg .v3 J32 vpos
einen 3-Komponenten-Vektor aus 32-Bit-Gleitkommazahlen in dem Zustandsraum des virtuellen Registers je Befehlsfolge. Nachdem ein Vektor deklariert wurde, können seine Komponenten mit Hilfe von Suffixen identifiziert werden, zum Beispiel vpos.x, vpos.y, vpos.z. In einer Ausführungsform ist m = 2, 3 oder 4 zulässig, und Suffixe wie zum Beispiel (.x, .y, .z, .w), (.0, .1, .2, .3) oder (.r, .g, .b, .a) werden zum Identifizieren von Komponenten verwendet.

[0110] Da die Variablen virtuell sind, kann virtueller ISA-Code **410** jede beliebige Anzahl von Variablen in jedem der Zustandsräume definieren oder sich auf jede beliebige Anzahl von Variablen in jedem der Zustandsräume beziehen (außer .sreg, wobei die Variablen vorgegeben sind). Es ist möglich, dass die Anzahl von Variablen, die für einen bestimmten Zustandsraum in virtuellem ISA-Code **410** definiert sind, die Menge an Speicher des entsprechenden Typs in einer bestimmten Hardware-Implementierung überschreiten kann. Der virtuelle Befehlsübersetzer **412** ist vorteilhafterweise so konfiguriert, dass er geeignete Speicherverwaltungsbefehle enthält (zum Beispiel Bewegen von Daten zwischen Register und Off-Chip-Speicher), um Variablen bei Bedarf verfügbar zu machen. Der virtuelle Befehlsübersetzer **412** kann auch in der Lage sein, Fälle zu detektieren, wo eine temporäre Variable nicht mehr benötigt wird und ihr zugewiesener Raum zur Verwendung durch eine

andere Variable freigegeben wird. Es können herkömmliche Kompiliertechniken zum Zuweisen von Registern verwendet werden.

[0111] Obgleich die beispielhafte virtuelle ISA Vektorvariablentypen definiert, ist es des Weiteren nicht erforderlich, dass die Zielplattform Vektorvariablen unterstützt. Der virtuelle Befehlsübersetzer **412** kann jede beliebige Vektorvariable als eine Zusammenstellung einer zweckmäßigen Anzahl (zum Beispiel 2, 3 oder 4) von Skalaren implementieren.

5.3. Virtuelle Befehle

[0112] Die Fig. 8A–Fig. 8H sind Tabellen, die virtuelle Befehle auflisten, die in einer beispielhaften virtuellen ISA definiert sind. Ein Befehl wird anhand seiner Wirkung definiert, zum Beispiel Berechnen eines bestimmten Ergebnisses unter Verwendung eines oder mehrerer Operanden und Anordnen dieses Ergebnisses in einem Zielortregister, Einstellen eines Registerwertes und so weiter. Die meisten virtuellen Befehle sind typifiziert, um das Format von Eingaben und/oder Ausgaben zu identifizieren, und Aspekte der Befehlsausführung können vom Typ abhängen. Das allgemeine Format eines Befehls ist
Name.<Typ> Ergebnis, Operanden
wobei "Name" der Name des Befehls ist; ".<Typ>" ein Platzhalter für jeden der Typen ist, die in Fig. 6 aufgelistet sind; "Ergebnis" eine Variable ist, in der das Ergebnis gespeichert wird; und "Operanden" eine oder mehrere Variablen sind, die als Eingaben in den Befehl bereitgestellt werden. In einer Ausführungsform ist die virtuelle Architektur **300** ein Register-zu-Register-Prozessor, und "Ergebnis" und "Operanden" für andere Operationen als Speicherzugriffe (Fig. 8F) müssen Variablen in dem Zustandsraum des virtuellen Registers .reg (oder dem Zustandsraum des speziellen Registers .sreg im Fall einiger Operanden) sein.

[0113] Von einer Zielplattform wird erwartet, dass sie jeden der Befehle in der virtuellen ISA realisiert. Ein Befehl kann entweder als ein entsprechender Maschinenbefehl, der den spezifizierten Effekt hervorruft (im vorliegenden Text als "Hardware-Unterstützung" bezeichnet), oder als eine Abfolge von Maschinenbefehlen, die, wenn sie ausgeführt werden, den spezifizierten Effekt hervorrufen (im vorliegenden Text als "Software-Unterstützung" bezeichnet), realisiert werden. Der virtuelle Befehlsübersetzer **412** für eine bestimmte Zielplattform ist vorteilhafterweise dafür konfiguriert, den Maschinenbefehl oder die Maschinenbefehl-Abfolge entsprechend jedem virtuellen Befehl zu identifizieren.

[0114] Die folgenden Unterabschnitte beschreiben die verschiedenen Klassen von Befehlen, die in den Fig. 8A–Fig. 8H aufgelistet sind. Es versteht sich, dass die im vorliegenden Text vorgestellte Liste von Befehlen der Veranschaulichung dient und dass eine virtuelle ISA zusätzliche Befehle enthalten kann, die nicht ausdrücklich im vorliegenden Text beschrieben sind, und einige oder alle der im vorliegenden Text beschriebenen Befehle ausschließen kann.

5.3.1. Virtuelle Befehle – Arithmetik

[0115] Fig. 8A ist eine Tabelle **800**, die arithmetische Operationen auflistet, die in der beispielhaften virtuellen ISA definiert sind. In dieser Ausführungsform unterstützt die virtuelle Architektur nur Register-zu-Register-Arithmetik, und alle arithmetischen Operationen bearbeiten ein oder mehrere Operanden virtueller Register (in Fig. 8A als a, b, c dargestellt), um ein Ergebnis (d) hervorzubringen, das in ein virtuelles Register geschrieben wird. Somit befinden sich Operanden und Zielorte für arithmetische Operationen immer im Zustandsraum des virtuellen Registers .reg, außer dass die speziellen Register von Fig. 5 (im Zustandsraum des speziellen Registers .sreg) als Operanden verwendet werden können.

[0116] Die Liste der arithmetischen Operationen in Tabelle **800** enthält die vier arithmetischen Grundrechenarten: Addition (add), Subtraktion (sub), Multiplikation (mul) und Division (div). Diese Operationen können an allen ganzzahligen und Gleitkommadatentypen ausgeführt werden und erbringen ein Ergebnis des gleichen Typs wie die Eingaben. In einigen Ausführungsformen kann auch ein Rundungsmodusqualifikator zu dem Befehl hinzugefügt werden, um es dem Programmierer zu ermöglichen zu spezifizieren, wie das Ergebnis zu runden ist und ob im Fall ganzzahliger Operanden Sättigungsgrenzen auferlegt werden sollen.

[0117] Es werden auch drei zusammengesetzte arithmetische Operationen mit Operanden a, b, und c unterstützt: Multiplikation-Addition (mad), fusionierte Multiplikation-Addition (fma) und Summe der absoluten Differenz (sad). Multiplikation-Addition berechnet das Produkt a × b (mit Runden, durch Klammern angezeigt) und addiert c zu dem Ergebnis. Fusionierte Multiplikation-Addition unterscheidet sich von mad dadurch, dass das Produkt a × b nicht vor dem Addieren von c gerundet wird. Die Summe der absoluten Differenz berechnet den absoluten Wert |a – b| und addiert dann c.

[0118] Die restliche (rem) Operation wird nur an ganzzahligen Operanden ausgeführt und berechnet den Rest (a mod b), wenn der Operand a ist durch den Operanden b geteilt wird. Absoluter Wert (abs) und Negation (neg) sind einstellige Operationen, die in einem Gleitkomma- oder signierten ganzzahligen Format auf einen Operanden a angewendet werden können. Minimum-(min) und Maximum-(max)Operationen, die auf ganzzahlige oder Gleitkomma-Operanden angewendet werden können, setzen das

Zielortregister auf den kleineren Operanden oder größeren Operanden. Der Umgang mit Sonderfällen, in denen ein oder beide Operanden eine nicht-normale Zahl sind (zum Beispiel gemäß den IEEE 754-Standards), können ebenfalls spezifiziert werden.

[0119] Die übrigen Operationen in Tabelle **800** werden nur für Gleitkomma-Typen ausgeführt. Eine Bruch(frc)-Operation gibt den Bruchteil ihrer Eingabe als Ergebnis aus. Sinus (sin), Kosinus (cos) und Arkustangens des Verhältnisses (atan2) bilden zweckmäßige Befehle entsprechend trigonometrischen Funktionen. Basis-2-Logarithmus (lg2) und Potenzierung (ex2) werden ebenfalls unterstützt. Reziprokes (rep), Quadratwurzel (sqrt) und reziproke Quadratwurzel (rsqrt) werden ebenfalls unterstützt.

[0120] Es ist zu beachten, dass diese Liste von arithmetischen Operationen veranschaulichend ist und die Erfindung nicht einschränkt. Es könnten noch weitere Operationen oder Kombinationen von Operationen unterstützt werden, einschließlich jeglicher Operationen, von denen erwartet wird, dass sie mit genügender Häufigkeit aufgerufen werden.

[0121] In einigen Ausführungsformen definiert die virtuelle ISA auch Vektoroperationen. Fig. 8B ist eine Tabelle **810**, die Vektoroperationen auflistet, die durch eine beispielhafte virtuelle ISA unterstützt werden. Die Vektoroperationen enthalten eine Skalarprodukt(dot)-Operation, die das Skalarprodukt d der Operandenvektoren a und b berechnet; eine Kreuzprodukt(cross)-Operation, die das Vektor-Kreuzprodukt d der Operandenvektoren a und b berechnet; und eine Größenordnungs(mag)-Operation, welche die skalare Länge d eines Operandenvektors a berechnet. Die Vektorreduktions(vred)-Operation berechnet ein skalares Ergebnis d durch iteratives Ausführen der spezifizierten Operation <op> an den Elementen des Vektoroperanden a. In einer Ausführungsform werden nur die Reduktionsoperationen add, mul, min und max für Gleitkomma-Vektoren unterstützt. Für ganzzahlige Vektoren können auch zusätzliche Reduktionsoperationen (zum Beispiel und, oder und xoder, wie unten beschrieben) unterstützt werden.

[0122] Zusätzlich zu diesen Operationen können auch andere Vektoroperationen wie zum Beispiel Vektoraddition, Vektorskalierung und dergleichen (in Fig. 8B nicht angeführt) in der virtuellen ISA definiert werden.

[0123] Wie oben angemerkt, könnte es sein, dass einige Hardware-Realisierungen der virtuellen Architektur **300** keine Vektorverarbeitung unterstützen. Der virtuelle Befehlsübersetzer **412** für solche Realisierungen ist vorteilhafterweise dafür geeignet, zweckmäßige Abfolgen skalarer Maschinenbefehle zu erzeugen, um diese Operationen auszuführen. Der Fachmann ist in der Lage, zweckmäßige Abfolgen zu erstellen.

5.3.2 Virtuelle Befehle – Auswahl und Registereinstellung

[0124] Fig. 8C ist eine Tabelle **820**, die Auswahl- und Registereinstell-Operationen auflistet, die in der beispielhaften virtuellen ISA definiert werden. Diese Operationen, die an jedem beliebigen numerischen Datentyp ausgeführt können werden, stellen ein Zielortregister auf der Grundlage des Ergebnisses einer Vergleichsoperation ein. Die elementare Auswahl(sel)-Operation wählt den Operanden a, wenn c ungleich null ist, und den Operanden b, wenn c gleich null ist. Vergleichen und Einstellen (set) führt eine Vergleichsoperation <cmp> an den Operanden a und b aus, um ein Vergleichsergebnis t zu erzeugen, und setzt dann das Zielortregister d auf ein Boolesches wahr (~0) oder falsch (0), je nachdem, ob das Vergleichsergebnis t wahr (~0) oder falsch (0) ist. Die zulässigen Vergleichsoperationen <cmp> beinhalten in einer Ausführungsform gleich (t ist wahr, wenn a = b), größer als (t ist wahr, wenn a > b), kleiner als (t ist wahr, wenn a < b), größer-gleich (t ist wahr, wenn a ≥ b), kleiner-gleich (t ist wahr, wenn a ≤ b), und andere Vergleiche, die zum Beispiel beinhalten, ob a und/oder b numerische oder undefinierte Werte sind.

[0125] Die setb-Operation ist eine Variante des Vergleichen-und-Einstellens, die eine weitere Boolesche Operation <bop> zwischen dem Ergebnis t der Vergleichsoperation <cmp> und einem dritten Operanden c ausführt. Das Ergebnis der Booleschen Operation t <bop> c bestimmt, ob das Zielortregister d auf ein Boolesches wahr oder falsch gesetzt wird. Die zulässigen Booleschen Operationen <bop> beinhalten in einer Ausführungsform und, oder und xoder (siehe Fig. 8C, die unten beschrieben wird). Die setp-Operation ähnelt setb, außer dass zwei 1-Bit-"Prädikat"-Zielortregister eingestellt werden: Das Zielortregister d1 wird auf das Ergebnis von t <bop> c eingestellt, während das Zielortregister d2 auf das Ergebnis von (!t) <bop> c eingestellt wird.

5.3.3. Virtuelle Befehle – Logische und Bit-Manipulation

[0126] Fig. 8D ist eine Tabelle **830**, die logische und Bit-Manipulationsoperationen auflistet, die in der beispielhaften virtuellen ISA definiert sind. Die Bit-weisen Booleschen Operationen und, oder und xoder werden ausgeführt, indem die spezifizierte Operation an jedem Bit der Operanden a und b ausgeführt wird und das entsprechende Bit im Register d auf das Ergebnis eingestellt wird. Die Bit-weise Negations (not)-Operation invertiert jedes Bit des Operanden a, während die logische Negations(cnot)-Operation das Zielortregister auf 1 (Boolesches wahr) einstellt,

wenn a null ist (Boolesches falsch), und anderenfalls auf 0 (Boolesches falsch).

[0127] Bit-Verschiebungen werden durch Linksverschiebe(shl)- und Rechtsverschiebe(shr)-Operationen unterstützt, die das Bit-Feld im Operanden a um die Anzahl von Bits, die durch den Operanden b spezifiziert wird, nach links oder nach rechts verschieben. Für signierte Formate füllt die Rechtsverschiebung vorteilhafterweise vorangestellte Bits auf der Grundlage des Signier-Bits auf. Für unsignierte Formate füllt die Rechtsverschiebung vorangestellte Bits mit Nullen auf.

5.3.4. Virtuelle Befehle – Formatkonvertierung

[0128] Fig. 8E ist eine Tabelle **840**, die Formatkonvertierungsoperationen auflistet, die in der beispielhaften virtuellen ISA definiert sind. Der Formatkonvertierungs(cvt)-Befehl konvertiert einen Operanden a eines ersten Typs <aTyp> zu einem äquivalenten Wert in einem Zieltyp <dTyp> und speichert das Ergebnis im Zielortregister d. Gültige Typen in einer Ausführungsform sind in Fig. 6 aufgelistet. Nicht-typisierte Werte (.b<n>) können nicht in ganzzahlige oder Gleitkomma-Typen oder aus ganzzahligen oder Gleitkomma-Typen konvertiert werden. Eine Variante des Formatkonvertierungsbefehls gestattet es dem Programmierer, einen Rundungsmodus <mode> zu spezifizieren. Der Umgang mit Zahlen, die gesättigt werden, wenn sie als der Zieltyp ausgedrückt werden, können ebenfalls spezifiziert werden.

5.3.5. Virtuelle Befehle – Datenbewegung und gemeinsame Nutzung von Daten

[0129] Fig. 8F ist eine Tabelle **850**, die Datenbewegungs- und Datengemeinschaftsnutzungsbefehle auflistet, die in der beispielhaften virtuellen ISA definiert werden. Die Bewegungs(mov)-Operation setzt das Zielortregister d auf den Wert des unmittelbaren Operanden a oder, wenn der Operand a ein Register ist, auf den Inhalt des Registers a. Die Bewegungsoperation kann auf Zustandsräume vom virtuellen Register-Typ beschränkt werden, zum Beispiel .reg und .sreg in Fig. 7.

[0130] Der Lade(ld)-Befehl lädt einen Wert von einem Quellenort im Speicher in das Zielortregister d, das sich in einer Ausführungsform im Zustandsraum des virtuellen Registers .reg befinden muss. Der .<space>-Qualifikator spezifiziert den Zustandsraum des Quellenortes und kann auf adressierbare Zustandsräume in Fig. 7 beschränkt sein, zum Beispiel andere Räume als .reg und .sreg (wo stattdessen die Bewegungsoperation verwendet werden kann). Da die virtuelle Architektur **300** in dieser Ausführungsform ein Register-zu-Register-Prozessor ist, wird der Ladebefehl vorteilhafterweise verwendet, um Variablen aus adressierbaren Zustandsräumen in den Zustandsraum des virtuellen Registers .reg zu übertragen, so dass sie als Operanden verwendet werden können.

[0131] Der spezifische Quellenort wird unter Verwendung eines Quellenparameters <src> identifiziert, der auf verschiedene Weise definiert werden kann, um verschiedene Adressierungsmodi zu unterstützen. Zum Beispiel kann in einigen Ausführungsformen der Quellenparameter <src> eines von Folgenden sein: eine benannte adressierbare Variable, deren Wert in d gespeichert werden soll; ein Verweis auf ein Register, in dem sich die Quellenadresse befindet; ein Verweis auf ein Register, in dem sich eine Adresse befindet, die einem Versatzwert hinzugefügt werden soll (als ein unmittelbarer Operand übermittelt); oder eine unmittelbare absolute Adresse.

[0132] Gleichermaßen speichert die Speicher(st)-Operation den Wert in einem Quellenregister a an einem Speicherort, der durch den Zielortparameter <dst> identifiziert wird. Das Quellenregister a muss sich in einer Ausführungsform in dem .reg-Zustandsraum befinden. Der Zielort muss sich in einem beschreibbaren und adressierbaren Zustandsraum befinden (zum Beispiel .local, .global oder .shared in Fig. 7). Der Zielortparameter <dst> kann auf verschiedene Weise definiert werden, um verschiedene Adressierungsmodi zu unterstützen, ähnlich dem Quellenparameter <src> in dem Ladebefehl. Der Speicherbefehl kann zum Beispiel verwendet werden, um ein Operationsergebnis von einem Register zu einem adressierbaren Zustandsraum zu übertragen.

[0133] In Ausführungsformen, wo Textur- und Oberflächenzustandsräume bereitgestellt sind, können zusätzliche virtuelle Befehle verwendet werden, um aus dem Texturspeicherzustandsraum (tex) zu lesen und um aus dem Oberflächenspeicherzustandsraum zu lesen (suld) und in den Oberflächenspeicherzustandsraum zu schreiben (sust). Die Operanden (t, x, y) für einen Texturlesevorgang spezifizieren den Texturidentifikator (t) und die Koordinaten (x, y). Gleichermaßen spezifizieren die Operanden (s, x, y) für einen Oberflächenlese- oder -schreibvorgang den Oberflächenidentifikator (s) und die Koordinaten (x, y).

[0134] Eine CTA-Befehlsfolge kann mit anderen CTA-Befehlsfolgen durch gemeinsame Nutzung von Daten mit anderen CTA-Befehlsfolgen zusammenwirken. Um zum Beispiel Daten innerhalb einer CTA gemeinsam zu nutzen, können die CTA-Befehlsfolgen virtuelle Lade- und Speicherbefehle (sowie den unten beschriebenen Befehl für eine nicht unterbrechbare (atomic) Aktualisierung ”atom”) verwenden, um Daten in die virtuellen Zustandräume je CTA zu schreiben und Daten aus den virtuellen Zustandräumen je CTA zu lesen. So kann eine CTA-Be-

fehlsfolge Daten unter Verwendung eines st.shared-Befehls mit einer in geeigneter Weise definierten Zielortadresse in den .shared-Zustandsraum schreiben. Eine weitere CTA-Befehlsfolge innerhalb derselben CTA kann anschließend die Daten unter Verwendung derselben Adresse in einem ld.shared-Befehl lesen. Die unten beschriebenen Synchronisationsbefehle (zum Beispiel bar und membar) können verwendet werden, um die richtige Abfolge von Datengemeinschaftsnutzungsoperationen in CTA-Befehlsfolgen zu gewährleisten, zum Beispiel, dass eine Daten erzeugende CTA-Befehlsfolge die Daten schreibt, bevor einen Datenverbrauchende CTA-Befehlsfolge sie liest. Gleichermaßen können st.global- und ld.global-Befehle für das Zusammenwirken und die gemeinsame Nutzung von Daten zwischen CTA-Befehlsfolgen in derselben CTA, CTAs in demselben Gitter und/oder verschiedenen Gittern in derselben Anwendung verwendet werden.

5.3.6. Virtuelle Befehle – Programmsteuerung

[0135] Fig. 8G ist eine Tabelle **860**, die Programmsteuerungsoperationen auflistet, in der beispielhaften virtuellen ISA bereitgestellt werden. Diese Steuerungsoperationen, mit denen der Fachmann vertraut ist, ermöglichen es einem Programmierer, die Programmausführung umzulenken. Ein Abzweig (bra) lenkt den Programmfluss zu einem Zielort <target>. In einigen Ausführungsformen wird ein Abzweigziel definiert, indem eine alphanumerische Markierung (label) vor den Zielbefehl in dem virtuellen ISA-Code gesetzt wird und diese Markierung als der Zielidentifikator <target> eines Abzweigbefehls verwendet wird. Zum Beispiel identifiziert in einer Ausführungsform:
label: add.int32 d, vrl, vr2
den ”add”-Befehl als ein Abzweigziel mit der Markierung ”label”. Der Befehl
bra label
an einer anderen Stelle in dem Code lenkt die Ausführung des markierten Befehls um.

[0136] Die call- und return(ret)-Befehle unterstützen Funktions- und Subroutinen-Aufrufe; fname identifiziert die Funktion oder Subroutine. (In einer Ausführungsform ist eine ”Subroutine” einfach eine Funktion, deren Rückmeldungswert ignoriert wird.) Die Funktion fname kann unter Verwendung einer func-Anweisung deklariert werden, und virtueller ISA-Code, der die Funktion definiert, kann ebenfalls bereitgestellt werden. Geschwungene Klammern {} oder andere Gruppierungssymbole können verwendet werden, um einen Code, der eine Funktion oder Subroutine definiert, von einem anderem virtuellen ISA-Code abzutrennen.

[0137] Für Funktionen kann eine Parameterliste <rv> spezifiziert werden, um zu identifizieren, wo Rückmeldungswerte zu speichern sind. Sowohl für Funktionen als auch für Subroutinen werden Eingabeargumente in Argumentlisten <args> spezifiziert. Wenn ”call” ausgeführt wird, so wird die Adresse des nächsten Befehls gespeichert. Wenn ”ret” ausgeführt wird, so wird ein Abzweig zu der gespeicherten Adresse genommen.

[0138] Der ”exit”-Befehl bricht eine CTA-Befehlsfolge, die auf ihn trifft, ab. Der Unterbrechungsbefehl ruft eine Prozessor-definierte oder Benutzer-definierte Unterbrechungsroutine auf. Der Haltepunkt(brkpt)-Befehl setzt die Ausführung aus und ist zum Beispiel für Fehlerbeseitigungszwecke nützlich. Der Funktionslos(nop)-Befehl ist ein Befehl, der bei Ausführung keinen Effekt hat. Er kann zum Beispiel verwendet werden, um zu steuern, wie schnell eine nächste Operation ausgeführt werden kann.

5.3.7. Virtuelle Befehle – Parallele Befehlsfolgen

[0139] Fig. 8H ist eine Tabelle **870**, die explizit parallele virtuelle Befehle auflistet, die in der beispielhaften virtuellen ISA gemäß einer Ausführungsform der vorliegenden Erfindung bereitgestellt werden. Diese Befehle unterstützen das zusammenwirkende Befehlsfolgenverhalten, das für die CTA-Ausführung gewünscht wird, wie zum Beispiel das Austauschen von Daten zwischen CTA-Befehlsfolgen.

[0140] Der Sperr(bar)-Befehl zeigt an, dass eine CTA-Befehlsfolge, die ihn erreicht, vor dem Ausführen weiterer Befehle so lange warten muss, bis alle anderen CTA-Befehlsfolgen (in derselben CTA) ebenfalls denselben Sperrbefehl erreicht haben. Es kann jede beliebige Anzahl von Sperrbefehlen in einem CTA-Programm verwendet werden. In einer Ausführungsform benötigt der Sperrbefehl keine Parameter (unabhängig davon, wie viele Sperren verwendet werden), da alle CTA-Befehlsfolgen die n-te Sperre erreichen müssen, bevor eine Befehlsfolge zur (n + 1)-ten Sperre voranschreiten kann, und so weiter.

[0141] In anderen Ausführungsformen kann der Sperrbefehl parametrisiert werden, zum Beispiel durch Spezifizieren einer Anzahl von CTA-Befehlsfolgen (oder Identifikatoren bestimmter CTA-Befehlsfolgen), die an einer bestimmten Sperre warten müssen.

[0142] Wieder andere Ausführungsformen stellen sowohl ”Warte”- als auch ”Nicht-warte”-Sperrbefehle bereit. Bei einem Warte-Sperrbefehl wartet die CTA-Befehlsfolge, bis die anderen relevanten CTA-Befehlsfolgen ebenfalls die Sperre erreicht haben. Bei einem Nicht-warte-Befehl zeigt die CTA-Befehlsfolge an, dass sie angekommen ist, aber sie kann fortgesetzt werden, bevor andere CTA-Befehlsfolgen eintreffen. An einer bestimmten Sperre können einige CTA-Befehlsfolgen warten, während an-

dere nicht warten.

[0143] In einigen Ausführungsformen kann der virtuelle bar-Befehl verwendet werden, um CTA-Befehlsfolgen zu synchronisieren, die zusammenwirken oder Daten unter Verwendung von Zustandsräumen gemeinsam genutzten Speichers gemeinsam nutzen. Nehmen wir zum Beispiel an, dass ein Satz von CTA-Befehlsfolgen (der einige oder alle Befehlsfolgen der CTA enthalten kann) jeweils einige Daten in einer Variable je Befehlsfolge erzeugt (zum Beispiel eine Variable ”myData” eines virtuellen .fp32-Registers) und dann die Daten liest, die durch eine andere CTA-Befehlsfolge in dem Satz erzeugt werden. Die Abfolge von Befehlen:
st.shared.fp32 myWriteAddress, myData; bar;
ld.shared.fp32 myData, myReadAddress;
wobei myWriteAddress und myReadAddress Variablen je Befehlsfolge sind, die Adressen in dem .shared-Zustandsraum entsprechen, sorgt für das gewünschte Verhalten. Nachdem jede CTA-Befehlsfolge ihre erzeugten Daten in den gemeinsam genutzten Speicher geschrieben hat, wartet sie, bis alle CTA-Befehlsfolgen ihre Daten gespeichert haben. Dann geht sie zum Lesen von Daten (die durch eine andere CTA-Befehlsfolge geschrieben worden sein können) aus dem gemeinsam genutzten Speicher über.

[0144] Der Speichersperr(membar)-Befehl zeigt an, dass jede CTA-Befehlsfolge zu warten hat, bis ihre zuvor angeforderten Speicheroperationen (oder mindestens alle Schreiboperationen) vollendet sind. Dieser Befehl garantiert, dass ein Speicherzugriff, der nach dem membar-Befehl erfolgt, das Ergebnis aller vor ihm erfolgten Schreiboperationen sieht. Der membar-Befehl verwendet in einer Ausführungsform einen optionalen Zustandsraum-Namen <space>, um seine Reichweite auf Speicheroperationen zu beschränken, die sich auf den spezifizierten Zustandsraum richten, der ein Speicherzustandsraum sein muss (zum Beispiel nicht die .reg- oder .sreg-Zustandsräume). Wenn kein Zustandsraum-Name spezifiziert ist, so wartet die CTA-Befehlsfolge, bis alle ausstehenden Operationen vollendet sind, die sich auf alle Speicherzustandsräume richten.

[0145] Der atomische-Aktualisierungs(atom)-Befehl veranlasst eine nicht unterbrechbare Aktualisierung (Lesen-Modifizieren-Schreiben) an einer gemeinsam genutzten Variable a, die durch einen Verweis <ref> identifiziert wird. Die gemeinsam genutzte Variable a kann sich in jedem beliebigen gemeinsam genutzten Zustandsraum befinden, und wie bei anderen Speicherverweisen können verschiedene Adressierungsmodi verwendet werden. Zum Beispiel kann <ref> eines von Folgenden sein: eine benannte adressierbare Variable a; ein Verweis auf ein Register, in dem sich die Adresse der Variable a befindet; ein Verweis auf ein Register, in dem sich eine Adresse befindet, die einem Versatzwert hinzugefügt werden soll (als ein unmittelbarer Operand übermittelt), um die Variable a zu lokalisieren; oder eine unmittelbare absolute Adresse der Variable a. Die CTA-Befehlsfolge lädt die Variable a von dem Ort des gemeinsam genutzten Zustandsraums in ein Zielortregister d und aktualisiert dann die Variable a unter Verwendung einer spezifizierten Operation <op>, die an einem Operanden a und (je nach der Operation) an einem zweiten und einem dritten Operanden b und c ausgeführt wird, wobei das Ergebnis an den Ort zurückgespeichert wird, der durch <ref> identifiziert wird. Das Zielortregister d behält den ursprünglich geladenen Wert von a. Die Lade-, Aktualisierungs- und Speicheroperationen werden nicht unterbrechbar ausgeführt, wodurch garantiert wird, dass keine andere CTA-Befehlsfolge auf die Variable a zugreift, während eine erste CTA-Befehlsfolge eine nicht unterbrechbare (atomic) Aktualisierung ausführt. In einer Ausführungsform ist die Variable a auf den .global- oder .shared-Zustandsraum beschränkt und kann in der gleichen Weise wie für die oben beschriebenen Lade- und Speicheroperationen spezifiziert werden.

[0146] In einigen Ausführungsformen brauchen nur bestimmte Operationen als nicht unterbrechbare Aktualisierungen ausgeführt zu werden. Zum Beispiel werden in einer Ausführungsform möglicherweise nur die folgenden Operationen <op> spezifiziert, wenn a vom Gleitkomma-Typ ist: Addieren von a zu b; Ersetzen von a durch das Minimum oder Maximum von a und b; und eine ternäre Vergleich-und-Tausch-Operation, die a durch c ersetzt, wenn a gleich b ist, und a ansonsten unverändert lässt. Für ein ganzzahliges a können zusätzliche Operationen unterstützt werden, zum Beispiel Bit-weises und, oder und xoder zwischen Operanden a und b sowie Inkrementieren oder Dekrementieren des Operanden a. Es könnten noch weitere nicht unterbrechbare Operationen oder Kombinationen von Operationen unterstützt werden.

[0147] Der vote-Befehl führt eine Reduktionsoperation <op> an einem Booleschen (zum Beispiel Typ .b1) Operanden a in einer vorgegebenen Gruppe von CTA-Befehlsfolgen aus. In einer Ausführungsform spezifiziert die virtuelle Architektur, dass CTA-Befehlsfolgen in SIMD-Gruppen ausgeführt werden und dass die vorgegebene Gruppe einer SIMD-Gruppe entspricht. In anderen Ausführungsformen können andere Gruppen von CTA-Befehlsfolgen durch die virtuelle Architektur oder den Programmierer definiert werden. Die Reduktionsoperation <op> bringt es mit sich, dass der Ergebniswert d auf der Basis der Reduktion des Operanden a in den CTA-Befehlsfolgen in der Gruppe und der durch den .<op>-Qualifikator spezifizierten Reduktionsoperation auf einen Booleschen Wahr- oder Falsch-Zustand eingestellt wird. In einer Ausführungsform sind die zulässigen Reduktionsoperationen: (1) .all, wobei d wahr ist, wenn a für

alle CTA-Befehlsfolgen in der Gruppe wahr und ansonsten falsch ist; (2) .any, wobei d wahr ist, wenn a für jede CTA-Befehlsfolge in der Gruppe wahr ist; und (3) .uni, wobei d wahr ist, wenn a für alle aktiven CTA-Befehlsfolgen in der Gruppe den gleichen Wert (entweder wahr oder falsch) hat.

5.3.8. Virtuelle Befehle – Bedingte Ausführung

[0148] In einigen Ausführungsformen unterstützt die virtuelle ISA die bedingte Ausführung jedes Befehls. Bei der bedingten Ausführung wird dem Befehl ein Boolescher "Schutzprädikat"-Wert zugeordnet, und der Befehl wird nur ausgeführt, wenn zum Zeitpunkt der Ausführung das Schutzprädikat als wahr beurteilt wird.

[0149] In der beispielhaften virtuellen ISA kann ein Schutzprädikat jede beliebige 1 Bit große Boolesche Variable eines virtuellen Registers sein (im vorliegenden Text mit P bezeichnet). Eine bedingte Ausführung wird durch Ersetzen eines Prädikatschutzes @P oder eines Nicht-Prädikatschutzes @!P vor dem opcode eines Befehls angezeigt. Ein Wert wird in dem Prädikatregister festgesetzt, zum Beispiel durch Identifizieren von P als das Zielortregister für einen Befehl, der ein Boolesches Ergebnis hervorbringt, wie zum Beispiel den setp-Befehl in der Tabelle **820** (Fig. 8C). Bei Antreffen des Schutzprädikats @P oder @!P liest der virtuelle Prozessor das P-Register. Für den Schutz @P wird, wenn P wahr ist, der Befehl ausgeführt; wenn nicht, so wird er übersprungen. Für den Schutz @!P wird der Befehl ausgeführt, wenn P falsch ist, und anderenfalls übersprungen. Das Prädikat P wird zum Ausführungszeitpunkt für jede CTA-Befehlsfolge beurteilt, die auf den bedingten Befehl trifft. Somit könnten einige CTA-Befehlsfolgen einen bedingten Befehl ausführen, während andere CTA-Befehlsfolgen dies nicht tun.

[0150] In einigen Ausführungsformen können Prädikate gesetzt werden, während Befehle ausgeführt werden. Zum Beispiel können bestimmte der virtuellen Befehle in den Tabellen **800**–**870** (Fig. 8A–Fig. 8H) einen Parameter entgegennehmen, der ein Prädikatregister als eine Ausgabe spezifiziert. Solche Befehle aktualisieren das spezifizierte Prädikatregister auf der Grundlage einer Eigenschaft des Befehlsergebnisses. Zum Beispiel könnte ein Prädikatregister verwendet werden, um anzuzeigen, ob das Ergebnis einer arithmetischen Operation eine spezielle Zahl (zum Beispiel null, unendlich oder keine Zahl in Gleitkomma-Operationen nach IEEE 754) ist, und so weiter.

6. Virtueller Befehlsübersetzer

[0151] Wie oben mit Bezug auf Fig. 4 angemerkt, richtet sich ein virtueller Befehlsübersetzer **412** auf eine bestimmte Plattformarchitektur. Der virtuelle Befehlsübersetzer **412**, der zum Beispiel als ein Software-Programm implementiert werden könnte, das auf einem Prozessor wie zum Beispiel der CPU **102** von Fig. 1 ausgeführt wird, empfängt einen virtuellen ISA-Code **410** und übersetzt ihn in Ziel-ISA-Code **414**, der auf der bestimmten Plattformarchitektur ausgeführt werden kann, auf die sich der virtuelle Befehlsübersetzer **412** richtet (zum Beispiel durch die PPU **122** von Fig. 1). Der virtuelle Befehlsübersetzer **412** bildet die virtuellen Variablen, die in dem virtuellen ISA-Code **410** deklariert werden, auf verfügbare Speicherorte ab, einschließlich Prozessorregister, On-Chip-Speicher, Off-Chip-Speicher und so weiter. In einigen Ausführungsformen bildet der virtuelle Befehlsübersetzer **412** jeden der virtuellen Zustandräume auf einen bestimmten Speichertyp ab. Zum Beispiel kann der .reg-Zustandsraum auf Befehlsfolge-spezifische Datenregister abgebildet werden, der .shared-Zustandsraum auf gemeinsam nutzbaren Speicher des Prozessors, der .global-Zustandsraum auf eine Region des virtuellen Speichers, die dem Anwendungsprogramm zugewiesen ist, und so weiter. Es sind noch weitere Abbildungen möglich.

[0152] Die virtuellen Befehle in dem virtuellen ISA-Code **410** werden in Maschinenbefehle übersetzt. In einer Ausführungsform ist der virtuelle Befehlsübersetzer **412** dafür konfiguriert, jeden virtuellen ISA-Befehl auf einen entsprechenden Maschinenbefehl oder eine entsprechende Abfolge von Maschinenbefehlen abzubilden, je nachdem, ob ein entsprechender Maschinenbefehl in dem Befehlssatz des Prozessors existiert, der die CTA-Befehlsfolgen ausführt.

[0153] Der virtuelle Befehlsübersetzer **412** bildet auch die CTA-Befehlsfolgen auf "physische" Befehlsfolgen oder Prozesse in der Zielplattformarchitektur ab. Wenn zum Beispiel die Zielplattformarchitektur mindestens n_0 gleichzeitige Befehlsfolgen unterstützt, so kann jede CTA-Befehlsfolge auf eine physische Befehlsfolge abgebildet werden, und der virtuelle Befehlsübersetzer **412** kann einen virtuellen Befehlscode für eine einzelne CTA-Befehlsfolge mit der Erwartung erzeugen, dass die Zielplattform **440** den Code für n_0 Befehlsfolgen mit n_0 eindeutige Identifikatoren ausführt. Wenn die Zielplattformarchitektur weniger als n_0 Befehlsfolgen unterstützt, so kann der virtuelle Befehlsübersetzer **412** virtuellen ISA-Code **410**, der Befehle enthält, die mehreren CTA-Befehlsfolgen entsprechen, mit der Erwartung erzeugen, dass dieser Code einmal je CTA ausgeführt wird, wodurch mehrere CTA-Befehlsfolgen auf eine einzelne physische Befehlsfolge oder einen einzelnen physischen Prozess abgebildet werden.

[0154] Insbesondere werden virtuelle Befehle, die sich auf eine gemeinsame Datennutzung beziehen (zum Beispiel Last-, Speicher- und nicht unterbrechbare(atomic)-Aktualisierungs-Befehle, die auf

.shared- oder .global-Zustandsraum zugreifen), und/oder zusammenwirkendes Befehlsfolge-Verhalten (zum Beispiel Sperr-, atomische-Aktualisierungs- und andere Befehle in Fig. 8H) in Maschinenbefehle oder Abfolgen von Maschinenbefehle übersetzt. Zielplattformarchitekturen, die für eine CTA-Ausführung optimiert sind, enthalten vorteilhafterweise Hardware-unterstützte Sperrbefehle, zum Beispiel mit Zählern und/oder Registern in der Befehlseinheit zum Zählen der Anzahl von Befehlsfolgen, die an dem Sperrbefehl angekommen sind, und zum Setzen von Markierungen, die verhindern, dass weitere Befehle für eine Befehlsfolge ausgegeben werden, während die Befehlsfolge an einer Sperre wartet. Andere Zielarchitekturen bieten möglicherweise keine direkte Hardware-Unterstützung für eine Befehlsfolgensynchronisation, wobei in diesem Fall andere Techniken zur Kommunikation zwischen Befehlsfolgen (zum Beispiel Semaphoren, Statusgruppierungen im Speicher oder dergleichen) verwendet werden können, um das gewünschte Verhalten hervorzurufen.

[0155] Bedingte Befehle werden ebenfalls in Maschinenbefehle übersetzt. In einigen Fällen unterstützt die Ziel-Hardware direkt eine bedingte Ausführung. In anderen Fällen können Prädikate gespeichert werden, zum Beispiel in Prozessorregistern, wobei bedingte Abzweigbefehle oder dergleichen verwendet werden, um die Register abzufragen und das gewünschte Laufzeitverhalten hervorzurufen, indem bedingte Befehle bedingt umher verzweigt werden.

[0156] Fig. 9 ist ein Flussdiagramm eines Prozesses **900** zur Verwendung eines virtuellen Befehlsübersetzers gemäß einer Ausführungsform der vorliegenden Erfindung. Bei Schritt **902** schreibt ein Programmierer CTA-Programmcode in einer höheren Sprache. In einer Ausführungsform definiert der CTA-Programmcode das gewünschte Verhalten einer einzelnen CTA-Befehlsfolge und kann die Befehlsfolge-ID (einschließlich der CTA-ID und/oder Gitter-ID) als einen Parameter verwenden, um Aspekte des Verhaltens der CTA-Befehlsfolge zu definieren oder zu steuern. Zum Beispiel kann ein Ort in einem gemeinsam genutzten Speicher, der zu lesen oder zu beschreiben ist, als eine Funktion der Befehlsfolge-ID bestimmt werden, so dass verschiedene CTA-Befehlsfolgen in derselben CTA aus verschiedenen Speicherorten in dem gemeinsam genutzten Speicher lesen und/oder in verschiedene Speicherorte in dem gemeinsam genutzten Speicher schreiben. In einer Ausführungsform ist CTA-Programmcode als Teil von einem Anwendungsprogrammcode enthalten (zum Beispiel Programmcode **402** von Fig. 4). Zusätzlich zum Definieren des Verhaltens von CTA-Befehlsfolgen kann der Anwendungsprogrammcode auch CTAs und/oder Gitter, Einricht-Eingabe- und -Ausgabedatensätze usw. definieren.

[0157] Bei Schritt **904** erzeugt ein Kompilierer (zum Beispiel der Kompilierer **408** von Fig. 4) einen virtuellen ISA-Code, der das Verhalten einer einzelnen (virtuellen) CTA-Befehlsfolge definiert, aus dem höhersprachigen Code. Wenn der Code sowohl CTA-Programmcode als auch anderen Code enthält, so kann der Kompilierer **408** den CTA-Programmcode von dem übrigen Code trennen, so dass nur der CTA-Programmcode verwendet wird, um virtuellen ISA-Code zu erzeugen. Es können herkömmliche Techniken zum Kompilieren von Programmcode, der in einer Sprache geschrieben wurde, in eine andere (virtuelle) Sprache verwendet werden. Es ist anzumerken, dass, da der erzeugte Code in einer virtuellen Sprache vorliegt, der Kompilierer nicht an eine bestimmte Hardware gebunden oder für eine bestimmte Hardware optimiert zu werden braucht. Der Kompilierer kann den virtuellen ISA-Code optimieren, der aus einer bestimmten Abfolge von einem eingegebenem Code erzeugt wurde (so dass zum Beispiel kürzere Abfolgen von virtuellen ISA-Befehlen bevorzugt werden). Programmcode in der virtuellen ISA kann im Speicher auf einer Festplatte gespeichert und/oder an eine große Vielzahl verschiedener Plattformarchitekturen verteilt werden, einschließlich Architekturen, die physisch anders als die virtuelle Architektur **300** von Fig. 3 aufgebaut sind. Der Code in der virtuellen ISA ist maschinenunabhängig und kann auf jeder Zielplattform ausgeführt werden, für die ein virtueller Befehlsübersetzer verfügbar ist. In alternativen Ausführungsformen kann ein Programmierer CTA-Programmcode direkt in die virtuelle ISA schreiben, oder virtueller ISA-Code kann durch ein Programm automatisch erzeugt werden. Wenn der Programmcode anfänglich als virtueller ISA-Code erzeugt wird, so kann der Kompilierungsschritt **904** weggelassen werden.

[0158] Bei Schritt **906** liest ein virtueller Befehlsübersetzer (zum Beispiel der Übersetzer **412** von Fig. 4) den virtuellen ISA-Code und erzeugt Code in einer Ziel-ISA, der auf einer Zielplattform ausgeführt werden kann. Im Gegensatz zu dem Kompilierer richtet sich der virtuelle Befehlsübersetzer auf eine bestimmte (reale) Plattformarchitektur und ist vorteilhafterweise so konfiguriert, den Ziel-ISA-Code für die beste Leistung auf dieser Architektur anzupassen und zu optimieren. In einer Ausführungsform, wo die Zielarchitektur mindestens n_0 Befehlsfolgen unterstützt, erzeugt der virtuelle Befehlsübersetzer ein Zielbefehlsfolgenprogramm, das gleichzeitig durch jede von no Befehlsfolgen ausgeführt werden kann, um eine CTA zu realisieren. In einer weiteren Ausführungsform erzeugt der virtuelle Befehlsübersetzer ein Zielprogramm, das Software-Techniken (zum Beispiel Befehlsabfolgen) verwendet, um n_0 gleichzeitige Befehlsfolgen zu emulieren, von denen jede Befehle ausführt, die dem virtuellen ISA-Code entsprechen. Der Übersetzer kann zum Zeitpunkt der Programminstallation, während der Programminitialisie-

rung oder an genau festgelegten Zeitpunkten während der Programmausführung aktiv sein.

[0159] Bei Schritt **908** führt ein Prozessor auf der Zielplattform (zum Beispiel die PPU **122** von Fig. 1) den Ziel-ISA-Code aus, um Daten zu verarbeiten. In einigen Ausführungsformen kann der Schritt **908** enthalten, Befehle und Zustandsparameter in den Prozessor einzuspeisen, um sein Verhalten zu steuern, wie weiter unten noch beschrieben wird.

[0160] Es versteht sich, dass der Prozess **900** veranschaulichend ist und dass Variationen und Modifikationen möglich sind. Schritte, die als sequenziell beschrieben sind, können parallel ausgeführt werden, die Reihenfolge der Schritte kann variiert werden, und Schritte können modifiziert oder kombiniert werden. Zum Beispiel kann in einigen Ausführungsformen ein Programmierer CTA-Programmcode unter Verwendung der virtuellen ISA direkt schreiben, wodurch die Notwendigkeit eines Kompilierers entfällt, der virtuellen ISA-Code erzeugt. In anderen Ausführungsformen wird der CTA-Programmcode als Teil eines großen Anwendungsprogramms geschrieben, das zum Beispiel auch Code enthält, der die Dimensionen einer CTA und/oder eines Gitters aus CTAs definiert, die ausgeführt werden sollen, um ein bestimmtes Problem zu lösen. In einer Ausführungsform werden nur jene Abschnitte des Codes, die das CTA-Programm darstellen, in virtuellen ISA-Code kompiliert. Andere Abschnitte können in andere (reale oder virtuelle) Befehlssätze kompiliert werden.

[0161] In anderen Ausführungsformen kann ein einzelner virtueller Befehlsübersetzer dafür konfiguriert sein, mehrere Versionen des Zielcodes zu erzeugen, die für verschiedene Zielplattformen geeignet sind. Zum Beispiel könnte der Übersetzer einen Programmcode in einer höheren Sprache (zum Beispiel C), Maschinencode für eine PPU und/oder Maschinencode für eine Einzelkern- oder Mehrkern-CPU, der ein PPU-Verhalten emuliert, unter Verwendung von Software-Techniken erzeugen.

7. Virtueller Ausführungstreiber

[0162] In einigen Ausführungsformen werden der virtuelle ISA-Code **410** und der virtuelle Befehlsübersetzer **412** dafür verwendet, den CTA-Programmcode zu erzeugen, der für jede Befehlsfolge einer CTA ausgeführt werden soll. Im Hinblick auf das Programmiermodell der Fig. 2A–Fig. 2B definiert das Spezifizieren des CTA-Programms eine Verarbeitungsaufgabe für jede CTA-Befehlsfolge **204**. Um das Modell zu vervollständigen, ist es auch notwendig, die Dimensionen einer CTA **202**, die Anzahl von CTAs in dem Gitter, den zu verarbeitenden Eingabedatensatz und so weiter zu definieren. Solche Informationen werden im vorliegenden Text als ”CTA-Steuerungsinformationen” bezeichnet.

[0163] Wie in Fig. 4 gezeigt, spezifiziert in einigen Ausführungsformen das Anwendungsprogramm **402** CTA-Steuerungsinformationen durch Verwenden von Rufen an Funktionen in einer virtuellen Bibliothek **404**. In einer Ausführungsform enthält die virtuelle Bibliothek **404** verschiedene Funktionsaufrufe, über die ein Programmierer eine CTA oder ein Gitter aus CTAs definieren und angeben kann, wann die Ausführung beginnen soll.

[0164] Fig. 10 ist eine Tabelle **1000**, die Funktionen auflistet, die in einer beispielhaften virtuellen Bibliothek **404** verfügbar sind. Die erste Gruppe von Funktionen bezieht sich auf das Definieren einer CTA. Genauer gesagt, ist die initCTA-Funktion die erste Funktion, die aufgerufen wird, um eine neue CTA zu erzeugen. Diese Funktion gestattet es dem Programmierer, die Dimensionen (ntid.x, ntid.y, ntid.z) einer CTA zu definieren und der neuen CTA einen Identifikator cname zuzuweisen. Die setCTAProgram-Funktion spezifiziert ein CTA-Programm, das durch jede Befehlsfolge des CTA-cname ausgeführt werden soll. Der Parameter pname ist ein logischer Programmidentifikator, der dem gewünschten CTA-Programm entspricht (zum Beispiel einem Programm in virtuellem ISA-Code). Die setCTAInputArray-Funktion gestattet es dem Programmierer, einen Quellenort (Startadresse und Größe) im globalen Speicher zu spezifizieren, von wo aus der CTA-cname Eingabedaten liest; und die setCTAOutputArray-Funktion gestattet es dem Programmierer, einen Zielort (Startadresse und Größe) im globalen Speicher zu spezifizieren, an den der CTA-cname Ausgabedaten schreibt. Die setCTAParams-Funktion wird verwendet, um Laufzeitkonstantenparameter für den CTA-cname einzustellen. Der Programmierer stellt der Funktion die Liste der Parameter – zum Beispiel als (Name, Wert)-Paare – zur Verfügung.

[0165] In einer Ausführungsform kann die setCTA-Params-Funktion auch durch den Kompilierer **408** verwendet werden, wenn er einen virtuellen ISA-Code **410** erzeugt. Da die setCTAParams-Funktion die Laufzeitparameter für die CTA definiert, kann der Kompilierer **408** diese Funktion so interpretieren, dass jeder Parameter als eine virtuelle Variable in dem .param-Zustandsraum definiert wird.

[0166] Die Tabelle **1000** listet auch Funktionen auf, die mit dem Definieren von Gittern aus CTAs zu tun haben. Die initGrid-Funktion ist die erste Funktion, die aufgerufen wird, um ein neues Gitter zu erzeugen. Diese Funktion gestattet es dem Programmierer, die Dimensionen (nctaid.x, nctaid.y, nctaid.z) eines Gitters zu definieren, den CTA-cname zu identifizieren, der in dem Gitter ausgeführt wird, und dem neu definierten Gitter einen Identifikator gname zuzuweisen. Die setGridInputArray- und die setGridOutputArray-Funktion ähneln den Funktionen auf CTA-Ebene und ermöglichen es, eine einzelne Ein-

gabe- und/oder Ausgabe-Gruppierung für alle Befehlsfolgen aller CTAs in einem Gitter zu definieren. Die setGridParams-Funktion wird dafür verwendet, Laufzeitkonstantenparameter für alle CTAs in dem Gitter gname einzustellen. Der Kompilierer **408** kann diese Funktion so interpretieren, dass jeder Parameter als eine virtuelle Variable in dem .const-Zustandsraum definiert wird.

[0167] Die launchCTA- und die launchGrid-Funktion signalisieren, dass die Ausführung des spezifizierten CTA-cname oder Gitter-gname beginnen soll.

[0168] Die virtuelle API kann auch andere Funktionen enthalten. Zum Beispiel bieten einige Ausführungsformen Synchronisationsfunktionen, die dafür verwendet werden können, die Ausführung mehrerer CTAs zu koordinieren. Wenn zum Beispiel die Ausgabe einer ersten CTA (oder eines ersten Gitters) als die Eingabe einer zweiten CTA (oder eines zweiten Gitters) verwendet werden soll, so kann die API eine Funktion (oder einen Parameter für die Startfunktion) enthalten, über die der virtuelle Ausführungstreiber angewiesen werden kann, dass die zweite CTA (oder das zweite Gitter) erst gestartet werden darf, wenn die Ausführung der ersten CTA (oder des ersten Gitters) vollendet ist.

[0169] Gemäß einer Ausführungsform der vorliegenden Erfindung können beliebige oder alle der Funktionsaufrufe in der Tabelle **1000** in ein Anwendungsprogramm aufgenommen werden, das auch das CTA-Programm (oder die CTA-Programme, wenn es mehrere CTAs in der Anwendung gibt) definiert, das auszuführen ist. Zum Kompilierungszeitpunkt werden die Funktionsaufrufe als Rufe an eine Anwendungsprogrammschnittstelle(Application Program Interface – API)-Bibliothek **404** behandelt, wodurch virtueller API-Code **406** erzeugt wird.

[0170] Der virtuelle API-Code wird unter Verwendung eines virtuellen Ausführungstreibers **418** realisiert, der jede Funktion in der virtuellen Bibliothek implementiert. In einer Ausführungsform ist der virtuelle Ausführungstreiber **418** ein Treiberprogramm, das in der CPU **102** von Fig. 1 ausgeführt wird und die PPU **122** steuert, welche die CTA-Befehlsfolgen realisiert. Die verschiedenen Funktionsaufrufe in der Tabelle **1000** von Fig. 10 werden so implementiert, dass sie dazu führen, dass der Treiber Befehle über einen Einspeicherungspuffer in der PPU **122** ausgibt. In einer weiteren Ausführungsform führt eine CPU ein oder mehrere Programme aus, um eine CTA zu realisieren, und der virtuelle Ausführungstreiber **418** stellt Parameter ein und steuert die Ausführung solcher Programme durch die CPU.

[0171] Es versteht sich, dass die im vorliegenden Text beschriebene virtuelle API veranschaulichend ist und dass Variationen und Modifikationen möglich sind. Es können auch andere Funktionen oder Kombinationen von Funktionen unterstützt werden. Techniken für virtuelle API, die dem Fachmann bekannt sind, können für die Zwecke der vorliegenden Erfindung angepasst werden.

Weitere Ausführungsformen

[0172] Obgleich die Erfindung anhand konkreter Ausführungsformen beschrieben wurde, erkennt der Fachmann, dass zahlreiche Modifikationen möglich sind. Zum Beispiel sind die konkrete virtuelle Architektur, die konkreten virtuellen Befehle und die virtuellen API-Funktionen, die im vorliegenden Text beschrieben sind, nicht erforderlich. An ihre Stelle können auch andere virtuelle Architekturen, Befehle und/oder Funktionen treten, die gleichzeitige, zusammenwirkende Befehlsfolgen unterstützen. Außerdem können sich die oben beschriebenen Ausführungsformen auf Fälle beziehen, wo alle Blöcke die gleiche Anzahl von Elementen haben, alle CTAs die gleiche Anzahl von Befehlsfolgen haben und dasselbe CTA-Programm ausführen, und so weiter. In einigen Anwendungen, zum Beispiel wo mehrere abhängige Gitter verwendet werden, kann es wünschenswert sein, CTAs in verschiedenen Gittern verschiedene CTA-Programme ausführen zu lassen oder verschiedene Anzahlen und/oder Größen von Gittern zu haben.

[0173] Obgleich im vorliegenden Text von ”zusammenwirkenden Befehlsfolgen-Gruppierungen” gesprochen wird, versteht es sich, dass einige Ausführungsformen Befehlsfolgen-Gruppierungen verwenden können, bei denen eine gemeinsame Datennutzung zwischen gleichzeitigen Befehlsfolgen nicht unterstützt wird. In anderen Ausführungsformen, in denen eine solche gemeinsame Datennutzung unterstützt wird, können die Befehlsfolgen, die für eine bestimmte Anwendung definiert sind, Daten gemeinsam nutzen, müssen es aber nicht.

[0174] Obgleich in den oben beschriebenen Ausführungsformen davon gesprochen werden kann, dass Befehlsfolge-Gruppierungen mehrere Befehlsfolgen haben, versteht es sich des Weiteren, dass in einem ”entarteten” Fall eine Befehlsfolge-Gruppierung auch nur eine einzige Befehlsfolge haben könnte. Somit könnte die vorliegende Erfindung dafür verwendet werden, eine Skalierbarkeit in Programmen bereitzustellen, die in einer CPU mit einem oder mehreren einfach-gereihten oder nebenläufigen Kernen ausgeführt werden sollen. Unter Verwendung der im vorliegenden Text beschriebenen Techniken könnte ein Programm in einer solchen Weise geschrieben werden, dass die Befehlsfolgen über eine beliebige Anzahl verfügbarer CPU-Kerne verteilt werden könnten (zum Beispiel unter Verwendung von Betriebssystem-Funktionalität), ohne dass eine Modifikation oder Rekompilierung des virtuellen ISA-Codes erfor-

derlich ist.

[0175] Die Begriffe ”virtuell” und ”real” werden im vorliegenden Text verwendet, um das Entkoppeln eines konzeptuellen Programmiermodells, das von einem Programmierer verwendet wird, um eine Problemlösung zu beschreiben, von einem echten Computersystem, auf dem das Programm letztendlich ausgeführt werden kann, widerzuspiegeln. Das ”virtuelle” Programmiermodell und seine zugehörige Architektur ermöglichen es einem Programmierer, eine höhere Sicht auf eine Parallelverarbeitungsaufgabe zu erlangen, und es versteht sich, dass es eventuell ein echtes Computersystem oder -gerät geben könnte, dessen Komponenten eins-zu-eins auf die im vorliegenden Text beschriebenen Komponenten der virtuellen Architektur abgebildet werden können. Der virtuelle Code, einschließlich virtuellem ISA-Code und virtuellem API-Code, wird vorteilhafterweise als Code in einer Sprache realisiert, die eins-zu-eins dem Befehlssatz eines echten Verarbeitungsgerätes entsprechen kann, aber nicht muss. Wie aller Programmcode kann der im vorliegenden Text angesprochene virtuelle Code auf einem greifbaren Medium (zum Beispiel einem Hauptspeicher oder einer Festplatte) gespeichert werden, über ein Netzwerk übertragen werden, und so weiter.

[0176] Computerprogramme, die verschiedene Merkmale der vorliegenden Erfindung enthalten – einschließlich beispielsweise virtuellen ISA- und/oder virtuellen API-Code, virtuelle Befehlsübersetzer, virtuelle Treiber, Kompilierer, Bibliotheken virtueller Funktionen und dergleichen –, können auf verschiedenen computerlesbaren Medien zum Speichern und/oder Übertragen codiert werden. Zu geeigneten Medien gehören magnetische Platten oder Magnetband, optische Speichermedien wie zum Beispiel Compact-Disk (CD) oder DVD (Digital Versatile-Disk), Flashspeicher und dergleichen. Solche Programme können auch codiert und unter Verwendung von Trägersignalen übertragen werden, die für eine Übertragung über drahtgebundene, optische und/oder Drahtlos-Netze geeignet sind, die mit einer Vielzahl verschiedener Protokolle, einschließlich dem Internet, kompatibel sind. Computerlesbare Speichermedien, die mit dem Programmcode codiert sind, können mit einem kompatiblen Gerät gebündelt werden, oder der Programmcode kann separat von anderen Geräten bereitgestellt werden (zum Beispiel über einen Download aus dem Internet).

[0177] Des Weiteren können bestimmte Aktionen im vorliegenden Text so beschrieben werden, dass sie von einem ”Programmierer” unternommen werden. Es wird in Betracht gezogen, dass der Programmierer ein Mensch, ein automatisierter Prozess, der Programmcode mit allenfalls geringem menschlichen Eingreifen erzeugt, oder eine Kombination aus menschlicher Interaktion mit automatisierten oder teilweise automatisierten Prozessen zum Erzeugen von Programmcode sein kann.

[0178] Obgleich des Weiteren im vorliegenden Text beschriebene Ausführungsformen auf Merkmale bestimmter Zielplattformen Bezug nehmen können, ist die Erfindung nicht auf diese Plattformen beschränkt. Genau genommen, kann eine virtuelle Architektur in jeder beliebigen Kombination von Hardware- und/oder Software-Komponenten realisiert werden. Dem Fachmann ist klar, dass man davon ausgehen kann, dass verschiedene Realisierungen der gleichen virtuellen Architektur sich in der Effizienz und/oder im Durchsatz unterscheiden. Solche Unterschiede sind jedoch für die vorliegende Erfindung nicht von Bedeutung.

[0179] Obgleich also die Erfindung anhand konkreter Ausführungsformen beschrieben wurde, versteht es sich, dass die Erfindung alle Modifikationen und Äquivalente innerhalb des Geltungsbereichs der folgenden Ansprüche mit erfassen soll.

[0180] Die Erfindung weist des Weiteren die folgenden Konzepte auf:

Konzept 1 zum Definieren eines Parallelverarbeitungsvorgangs, das Konzept aufweisend: Bereitstellen von einem ersten Programmcode, der eine Abfolge von Operationen definiert, die für jede einer Mehrzahl von virtuellen Befehlsfolgen in einer Gruppierung zusammenwirkender virtueller Befehlsfolgen ausgeführt werden sollen; Kompilieren des ersten Programmcodes in ein Programm virtueller Befehlsfolgen, das eine Abfolge von Befehlen je Befehlsfolge definiert, die für eine repräsentative virtuelle Befehlsfolge der Mehrzahl von virtuellen Befehlsfolgen ausgeführt werden sollen, wobei die Abfolge von Befehlen je Befehlsfolge mindestens einen Befehl enthält, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Mehrzahl von virtuellen Befehlsfolgen definiert; und Speichern des Programms virtueller Befehlsfolgen.

Konzept 2 nach Konzept 1, ferner aufweisend: Übersetzen des gespeicherten Programms virtueller Befehlsfolgen in eine Abfolge von Befehlen, die mit einer Zielplattformarchitektur kompatibel sind.

Konzept 3 nach Konzept 1, ferner aufweisend: Bereitstellen von einem zweiten Programmcode, der eine Gruppierung zusammenwirkender virtueller Befehlsfolgen definiert, die dafür geeignet sind, einen Eingabedatensatz zu verarbeiten um einen Ausgabedatensatz zu erzeugen, wobei jede virtuelle Befehlsfolge in der Gruppierung gleichzeitig das Programm virtueller Befehlsfolgen ausführt; Konvertieren des zweiten Programmcodes in eine Abfolge von Funktionsaufrufen in einer Bibliothek virtueller Funktionen, wobei die Bibliothek

virtuelle Funktionen enthält, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen initialisieren und die Ausführung der Gruppierung zusammenwirkender virtueller Befehlsfolgen veranlassen; und Speichern der Abfolge von Funktionsaufrufen.

Konzept 4 nach Konzept 3, ferner aufweisend: Übersetzen des gespeicherten Programms virtueller Befehlsfolgen und der Abfolge von Funktionsaufrufen in einen Programmcode, der auf einer Zielplattformarchitektur ausgeführt werden kann, wobei der ausführbare Programmcode eine oder mehrere Plattformbefehlsfolgen definiert, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen ausführen.

Konzept 5 nach Konzept 4, ferner aufweisend: Ausführen des ausführbaren Programmcodes auf einem Computersystem, das mit der Zielplattformarchitektur kompatibel ist, wodurch der Ausgabedatensatz erzeugt wird; und Speichern des Ausgabedatensatzes in einem Speichermedium.

Konzept 6 nach Konzept 1, wobei die Abfolge von Befehlen je Befehlsfolge einen Befehl enthält, die Ausführung von Operationen für die repräsentative virtuelle Befehlsfolge an einen bestimmten Punkt in der Abfolge so lange auszusetzen, bis eine oder mehrere der anderen virtuellen Befehlsfolgen diesen bestimmten Punkt erreichen.

Konzept 7 nach Konzept 1, wobei die Abfolge von Befehlen je Befehlsfolge einen Befehl für die repräsentative virtuelle Befehlsfolge enthält, Daten in einem gemeinsam genutzten Speicher zu speichern, auf den eine oder mehrere der anderen virtuellen Befehlsfolgen Zugriff haben.

Konzept 8 nach Konzept 1, wobei die Abfolge von Befehlen je Befehlsfolge einen Befehl für die repräsentative virtuelle Befehlsfolge enthält, nicht unterbrechbar Daten zu lesen und zu aktualisieren, die in einem gemeinsam genutzten Speicher gespeichert sind, auf den eine oder mehrere der anderen virtuellen Befehlsfolgen Zugriff haben.

Konzept 9 nach Konzept 1, wobei das Programm virtueller Befehlsfolgen eine Variablendefinitionsaussage enthält, die eine Variable in einem aus einer Mehrzahl von virtuellen Zustandsräumen definiert, wobei verschiedene der Mehrzahl von virtuellen Zustandräumen verschiedenen Modi gemeinsamer Datennutzung zwischen den virtuellen Befehlsfolgen entsprechen.

Konzept 10 nach Konzept 9, wobei die Modi der gemeinsamen Datennutzung einen nicht gemeinsam genutzten Modus je Befehlsfolge und einen global gemeinsam genutzten Modus enthalten.

Konzept 11 nach Konzept 9, wobei die Modi der gemeinsamen Datennutzung einen nicht gemeinsam genutzten Modus je Befehlsfolge, einen gemeinsam genutzten Modus innerhalb einer Gruppierung virtueller Befehlsfolgen und einen global gemeinsam genutzten Modus enthalten.

Konzept 12 nach Konzept 9, wobei die Modi der gemeinsamen Datennutzung einen nicht gemeinsam genutzten Modus je Befehlsfolge, einen gemeinsam genutzten Modus innerhalb einen Gruppierung virtueller Befehlsfolgen, einen gemeinsam genutzten Modus zwischen mehreren Gruppierungen virtueller Befehlsfolgen und einen global gemeinsam genutzten Modus enthalten.

Konzept 13 zum Betreiben eines Zielprozessors, das Konzept aufweisend: Bereitstellen von einem Eingabeprogrammcode, der einen ersten Abschnitt enthält, der eine Abfolge von Operationen definiert, die für jede einer Mehrzahl von virtuellen Befehlsfolgen in einer Gruppierung virtueller Befehlsfolgen auszuführen sind, die dafür geeignet sind, einen Eingabedatensatz zu verarbeiten um einen Ausgabedatensatz zu erzeugen, wobei der Eingabeprogrammcode ferner einen zweiten Abschnitt enthält, der eine Dimension der Gruppierung virtueller Befehlsfolgen definiert; Kompilieren des ersten Abschnitts des Eingabeprogrammcodes in ein Programm virtueller Befehlsfolgen, das eine Abfolge von Befehlen je Befehlsfolge definiert, die für eine repräsentative virtuelle Befehlsfolge der Mehrzahl von virtuellen Befehlsfolgen ausgeführt werden sollen, wobei die Abfolge von Befehlen je Befehlsfolge mindestens einen Befehl enthält, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Mehrzahl von virtuellen Befehlsfolgen definiert; Konvertieren des zweiten Abschnitts des Eingabeprogrammcodes in eine Abfolge von Funktionsaufrufen an eine Bibliothek virtueller Funktionen, wobei die Bibliothek virtuelle Funktionen enthält, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen initialisieren und die Ausführung der Gruppierung zusammenwirkender virtueller Befehlsfolgen veranlassen; Übersetzen des Programms virtueller Befehlsfolgen und der Abfolge von Funktionsaufrufen in einen Programmcode, der auf einer Zielplattformarchitektur ausgeführt werden kann, wobei der ausführbare Programmcode eine oder mehrere reale Befehlsfolgen definiert, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen ausführen; Ausführen des ausführbaren Programmcodes auf einem Computersystem, das mit der Zielplattformarchitektur kompatibel ist, wodurch der Ausgabedatensatz erzeugt wird; und Speichern des Ausgabedatensatzes auf einem Speichermedium.

Konzept 14 nach Konzept 13, wobei der zweite Abschnitt des Eingabeprogrammcodes einen Programmcode enthält, der zwei oder mehr Dimensionen für die Gruppierung virtueller Befehlsfolgen definiert.

Konzept 15 nach Konzept 14, wobei der zweite Abschnitt des Eingabeprogrammcodes ferner enthält: einen Funktionsaufruf, der eine oder mehrere Dimensionen eines Gitters aus Gruppierungen

virtueller Befehlsfolgen definiert, wobei jede Gruppierung in dem Gitter ausgeführt werden soll.

Konzept 16 nach Konzept 13, wobei die Zielplattformarchitektur einen Master-Prozessor und einen Koprozessor enthält und wobei die Aktion des Übersetzens Folgendes enthält: Übersetzen des Programms virtueller Befehlsfolgen in einen Programmcode, der parallel durch mehrere Befehlsfolgen, die in dem Koprozessor definiert sind, ausgeführt werden kann; und Übersetzen der Abfolge von Funktionsaufrufen in eine Abfolge von Aufrufen an ein Treiberprogramm für den Koprozessor, wobei das Treiberprogramm in dem Master-Prozessor ausgeführt wird.

Konzept 17 nach Konzept 13, wobei die Zielplattformarchitektur eine zentrale Verarbeitungseinheit (CPU) enthält und wobei die Aktion des Übersetzens Folgendes enthält: Übersetzen des Programms virtueller Befehlsfolgen und mindestens eines Abschnitts der Abfolge von Funktionsaufrufen in einen Zielprogrammcode, der die Gruppierung virtueller Befehlsfolgen unter Verwendung einer Anzahl von CPU-Befehlsfolgen ausführt, die kleiner als die Anzahl virtueller Befehlsfolgen ist.

Konzept 18 zum Betreiben eines Zielprozessors, das Konzept aufweisend: Erhalten eines Programms virtueller Befehlsfolgen, das eine Abfolge von Befehlen je Befehlsfolge definiert, die für eine repräsentative virtuelle Befehlsfolge aus einer Mehrzahl von virtuellen Befehlsfolgen in einer Gruppierung virtueller Befehlsfolgen ausgeführt werden sollen, die dafür geeignet sind, einen Eingabedatensatz zu verarbeiten um einen Ausgabedatensatz zu erzeugen, wobei die Abfolge von Befehlen je Befehlsfolge mindestens einen Befehl enthält, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Mehrzahl von virtuellen Befehlsfolgen definiert; Erhalten eines zusätzlichen Programmcodes, der Dimensionen der Gruppierung virtueller Befehlsfolgen definiert; Übersetzen des Programms virtueller Befehlsfolgen und des zusätzlichen Programmcodes in einen Programmcode, der auf der Zielplattformarchitektur ausgeführt werden kann, wobei der ausführbare Programmcode eine oder mehrere Plattformbefehlsfolgen definiert, welche die Gruppierung virtueller Befehlsfolgen ausführen; Ausführen des ausführbaren Programmcodes auf einem Computersystem, das mit der Zielplattformarchitektur kompatibel ist, wodurch der Ausgabedatensatz erzeugt wird, und Speichern des Ausgabedatensatzes in einem Speicher.

Konzept 19 nach Konzept 18, wobei die Aktion des Erhaltens des Programms virtueller Befehlsfolgen enthält: Empfangen von einem Quellprogrammcode, der in einer höheren Programmiersprache geschrieben ist; und Kompilieren des Quellprogrammcodes um das Programm virtueller Befehlsfolgen zu erzeugen.

Konzept 20 nach Konzept 18, wobei die Aktion des Erhaltens des Programms virtueller Befehlsfolgen enthält: Lesen des Programms virtueller Befehlsfolgen von einem Speichermedium.

Konzept 21 nach Konzept 18, wobei die Aktion des Erhaltens des Programms virtueller Befehlsfolgen enthält: Empfangen des Programms virtueller Befehlsfolgen von einem räumlich abgesetzten Computersystem über ein Netzwerk.

ZITATE ENTHALTEN IN DER BESCHREIBUNG

Diese Liste der vom Anmelder aufgeführten Dokumente wurde automatisiert erzeugt und ist ausschließlich zur besseren Information des Lesers aufgenommen. Die Liste ist nicht Bestandteil der deutschen Patent- bzw. Gebrauchsmusteranmeldung. Das DPMA übernimmt keinerlei Haftung für etwaige Fehler oder Auslassungen.

Zitierte Nicht-Patentliteratur

- IEEE 754-Standards **[0096]**
- IEEE 754-Standards **[0118]**
- IEEE 754 **[0150]**

Schutzansprüche

1. Parallelverarbeitungsarchitektur zum Definieren eines Parallelverarbeitungsvorgangs, wobei die Parallelverarbeitungsarchitektur einen Parallelprozessor und einen Speicher aufweist,
wobei der Speicher einen ersten Programmcode enthält, der eine Abfolge von Operationen definiert, die für jede einer Mehrzahl von virtuellen Befehlsfolgen in einer Gruppierung zusammenwirkender virtueller Befehlsfolgen ausgeführt werden sollen,
wobei der Parallelprozessor betreibbar ist, den ersten Programmcode in ein Programm virtueller Befehlsfolgen zu kompilieren, das eine Abfolge von Befehlen je Befehlsfolge definiert, die für eine repräsentative virtuelle Befehlsfolge der Mehrzahl von virtuellen Befehlsfolgen ausgeführt werden sollen, wobei die Abfolge von Befehlen je Befehlsfolge mindestens einen Befehl enthält, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Mehrzahl von virtuellen Befehlsfolgen definiert; und
wobei der Speicher das Programm virtueller Befehlsfolgen enthält.

2. Parallelverarbeitungsarchitektur nach Anspruch 1, wobei der Parallelprozessor betreibbar ist, das gespeicherte Programm virtueller Befehlsfolgen in eine Abfolge von Befehlen, die mit einer Zielplattformarchitektur kompatibel sind, zu übersetzen.

3. Parallelverarbeitungsarchitektur nach Anspruch 1, wobei der Speichereinen zweiten Programmcode enthält, der eine Gruppierung zusammenwirkender virtueller Befehlsfolgen definiert, die dafür geeignet sind, einen Eingabedatensatz zu verarbeiten um einen Ausgabedatensatz zu erzeugen, wobei jede virtuelle Befehlsfolge in der Gruppierung gleichzeitig das Programm virtueller Befehlsfolgen ausführt,
wobei der Parallelprozessor betreibbar ist, den zweiten Programmcode in eine Abfolge von Funktionsaufrufen in einer Bibliothek virtueller Funktionen zu konvertieren, wobei die Bibliothek virtuelle Funktionen enthält, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen initialisieren und die Ausführung der Gruppierung zusammenwirkender virtueller Befehlsfolgen veranlassen; und
wobei der Speicher die Abfolge von Funktionsaufrufen enthält.

4. Parallelverarbeitungsarchitektur nach Anspruch 3, wobei der Parallelprozessor ferner betreibbar ist, das gespeicherte Programm virtueller Befehlsfolgen und die Abfolge von Funktionsaufrufen in einen Programmcode zu übersetzen, der auf einer Zielplattformarchitektur ausgeführt werden kann, wobei der ausführbare Programmcode eine oder mehrere Plattformbefehlsfolgen definiert, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen ausführen.

5. Parallelverarbeitungsarchitektur nach Anspruch 4, wobei die Parallelverarbeitungsarchitektur ferner ferner ein Computersystem aufweist, das mit der Zielplattformarchitektur kompatibel ist, wobei das Computersystem betreibbar ist, den ausführbaren Programmcode auszuführen, wodurch der Ausgabedatensatz erzeugt wird, und um den Ausgabedatensatz in einem Speichermedium zu speichern.

6. Parallelverarbeitungsarchitektur nach Anspruch 1, wobei die Abfolge von Befehlen je Befehlsfolge einen Befehl enthält, die Ausführung von Operationen für die repräsentative virtuelle Befehlsfolge an einen bestimmten Punkt in der Abfolge so lange auszusetzen, bis eine oder mehrere der anderen virtuellen Befehlsfolgen diesen bestimmten Punkt erreichen.

7. Parallelverarbeitungsarchitektur nach Anspruch 1, wobei die Abfolge von Befehlen je Befehlsfolge einen Befehl für die repräsentative virtuelle Befehlsfolge enthält, Daten in einem gemeinsam genutzten Speicher zu speichern, auf den eine oder mehrere der anderen virtuellen Befehlsfolgen Zugriff haben.

8. Parallelverarbeitungsarchitektur nach Anspruch 1, wobei die Abfolge von Befehlen je Befehlsfolge einen Befehl für die repräsentative virtuelle Befehlsfolge enthält, nicht unterbrechbar Daten zu lesen und zu aktualisieren, die in einem gemeinsam genutzten Speicher gespeichert sind, auf den eine oder mehrere der anderen virtuellen Befehlsfolgen Zugriff haben.

9. Parallelverarbeitungsarchitektur nach Anspruch 1, wobei das Programm virtueller Befehlsfolgen eine Variablendefinitionsaussage enthält, die eine Variable in einem aus einer Mehrzahl von virtuellen Zustandsräumen definiert, wobei verschiedene der Mehrzahl von virtuellen Zustandräumen verschiedenen Modi gemeinsamer Datennutzung zwischen den virtuellen Befehlsfolgen entsprechen.

10. Parallelverarbeitungsarchitektur nach Anspruch 9, wobei die Modi der gemeinsamen Datennutzung einen nicht gemeinsam genutzten Modus je Befehlsfolge und einen global gemeinsam genutzten Modus enthalten.

11. Parallelverarbeitungsarchitektur nach Anspruch 9, wobei die Modi der gemeinsamen Datennutzung einen nicht gemeinsam genutzten Modus je Befehlsfolge, einen gemeinsam genutzten Modus innerhalb einer Gruppierung virtueller Befehlsfolgen und einen global gemeinsam genutzten Modus enthalten.

12. Parallelverarbeitungsarchitektur nach Anspruch 9, wobei die Modi der gemeinsamen Datennutzung einen nicht gemeinsam genutzten Modus je Befehlsfolge, einen gemeinsam genutzten Modus innerhalb einen Gruppierung virtueller Befehlsfolgen, einen gemeinsam genutzten Modus zwischen mehreren Gruppierungen virtueller Befehlsfolgen und einen global gemeinsam genutzten Modus enthalten.

13. Parallelverarbeitungsarchitektur zum Betreiben eines Zielprozessors,
wobei die Parallelverarbeitungsarchitektur einen Parallelprozessor, einen Speicher und ein Computersystem, das mit der Zielplattformarchitektur kompatibel ist, aufweist,
wobei der Speicher einen Eingabeprogrammcode enthält, der einen ersten Abschnitt enthält, der eine Abfolge von Operationen definiert, die für jede einer Mehrzahl von virtuellen Befehlsfolgen in einer Gruppierung virtueller Befehlsfolgen auszuführen sind, die dafür geeignet sind, einen Eingabedatensatz zu verarbeiten um einen Ausgabedatensatz zu erzeugen,
wobei der Eingabeprogrammcode ferner einen zweiten Abschnitt enthält, der eine Dimension der Gruppierung virtueller Befehlsfolgen definiert;
wobei der Parallelprozessor betreibbar ist, den ersten Abschnitt des Eingabeprogrammcodes in ein Programm virtueller Befehlsfolgen zu kompilieren, das eine Abfolge von Befehlen je Befehlsfolge definiert, die für eine repräsentative virtuelle Befehlsfolge der Mehrzahl von virtuellen Befehlsfolgen ausgeführt werden sollen, wobei die Abfolge von Befehlen je Befehlsfolge mindestens einen Befehl enthält, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Mehrzahl von virtuellen Befehlsfolgen definiert;
wobei der Parallelprozessor betreibbar ist, den zweiten Abschnitt des Eingabeprogrammcodes in eine Abfolge von Funktionsaufrufen an eine Bibliothek virtueller Funktionen zu konvertieren, wobei die Bibliothek virtuelle Funktionen enthält, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen initialisieren und die Ausführung der Gruppierung zusammenwirkender virtueller Befehlsfolgen veranlassen;
wobei der Parallelprozessor betreibbar ist, das Programm virtueller Befehlsfolgen und die Abfolge von Funktionsaufrufen in einen Programmcode zu übersetzen, der auf einer Zielplattformarchitektur ausgeführt werden kann, wobei der ausführbare Programmcode eine oder mehrere reale Befehlsfolgen definiert, welche die Gruppierung zusammenwirkender virtueller Befehlsfolgen ausführen;
wobei das Computersystem betreibbar ist, den ausführbaren Programmcode auszuführen, wodurch der Ausgabedatensatz erzeugt wird, und um den Ausgabedatensatz auf einem Speichermedium zu speichern.

14. Parallelverarbeitungsarchitektur nach Anspruch 13, wobei der zweite Abschnitt des Eingabeprogrammcodes einen Programmcode enthält, der zwei oder mehr Dimensionen für die Gruppierung virtueller Befehlsfolgen definiert.

15. Parallelverarbeitungsarchitektur nach Anspruch 14, wobei der zweite Abschnitt des Eingabeprogrammcodes ferner enthält:
einen Funktionsaufruf, der eine oder mehrere Dimensionen eines Gitters aus Gruppierungen virtueller Befehlsfolgen definiert, wobei jede Gruppierung in dem Gitter ausgeführt werden soll.

16. Parallelverarbeitungsarchitektur nach Anspruch 13, wobei die Zielplattformarchitektur einen Master-Prozessor und einen Koprozessor enthält und wobei der Parallelprozessor betreibbar ist, das Programm virtueller Befehlsfolgen in einen Programmcode zu übersetzen, der parallel durch mehrere Befehlsfolgen, die in dem Koprozessor definiert sind, ausgeführt werden kann, und die Abfolge von Funktionsaufrufen in eine Abfolge von Aufrufen an ein Treiberprogramm für den Koprozessor zu übersetzen, wobei das Treiberprogramm in dem Master-Prozessor ausgeführt wird.

17. Parallelverarbeitungsarchitektur nach Anspruch 13, wobei die Zielplattformarchitektur eine zentrale Verarbeitungseinheit (CPU) enthält und wobei der Parallelprozessor betreibbar ist, das Programm virtueller Befehlsfolgen und mindestens einen Abschnitt der Abfolge von Funktionsaufrufen in einen Zielprogrammcode zu übersetzen, der die Gruppierung virtueller Befehlsfolgen unter Verwendung einer Anzahl von CPU-Befehlsfolgen ausführt, die kleiner als die Anzahl virtueller Befehlsfolgen ist.

18. Parallelverarbeitungsarchitektur zum Betreiben eines Zielprozessors,
wobei die Parallelverarbeitungsarchitektur einen Parallelprozessor, einen Speicher und ein Computersystem, das mit der Zielplattform kompatibel ist, aufweist,
wobei die Parallelverarbeitungsarchitektur betreibbar ist, ein Programm virtueller Befehlsfolgen zu erhalten, das eine Abfolge von Befehlen je Befehlsfolge definiert, die für eine repräsentative virtuelle Befehlsfolge aus einer Mehrzahl von virtuellen Befehlsfolgen in einer Gruppierung virtueller Befehlsfolgen ausgeführt werden sollen, die dafür geeignet sind, einen Eingabedatensatz zu verarbeiten um einen Ausgabedatensatz zu erzeugen,
wobei die Abfolge von Befehlen je Befehlsfolge mindestens einen Befehl enthält, der ein Zusammenwirkungsverhalten zwischen der repräsentativen virtuellen Befehlsfolge und einer oder mehreren anderen virtuellen Befehlsfolgen der Mehrzahl von virtuellen Befehlsfolgen definiert;
wobei die Parallelverarbeitungsarchitektur betreibbar

ist, einen zusätzlichen Programmcode zu erhalten, der Dimensionen der Gruppierung virtueller Befehlsfolgen definiert;
wobei der Parallelprozessor betreibbar ist, das Programm virtueller Befehlsfolgen und den zusätzlichen Programmcode in einen Programmcode zu übersetzen, der auf der Zielplattformarchitektur ausgeführt werden kann, wobei der ausführbare Programmcode eine oder mehrere Plattformbefehlsfolgen definiert, welche die Gruppierung virtueller Befehlsfolgen ausführen;
wobei das Computersystem betreibbar ist, den ausführbaren Programmcode auszuführen, wodurch der Ausgabedatensatz erzeugt wird, und
wobei der Speicher den Ausgabedatensatz enthält.

19. Parallelverarbeitungsarchitektur nach Anspruch 18, wobei die Parallelverarbeitungsarchitektur betreibbar ist, einen Quellprogrammcode zu empfangen, der in einer höheren Programmiersprache geschrieben ist und den Quellprogrammcode zu kompilieren, um das Programm virtueller Befehlsfolgen zu erzeugen.

20. Parallelverarbeitungsarchitektur nach Anspruch 18, wobei die Parallelverarbeitungsarchitektur betreibbar ist, das Programm virtueller Befehlsfolgen von einem Speichermedium zu lesen.

21. Parallelverarbeitungsarchitektur nach Anspruch 18, wobei die Parallelverarbeitungsarchitektur betreibbar ist, das Programm virtueller Befehlsfolgen von einem räumlich abgesetzten Computersystem über ein Netzwerk zu empfangen.

Es folgen 10 Blatt Zeichnungen

Anhängende Zeichnungen

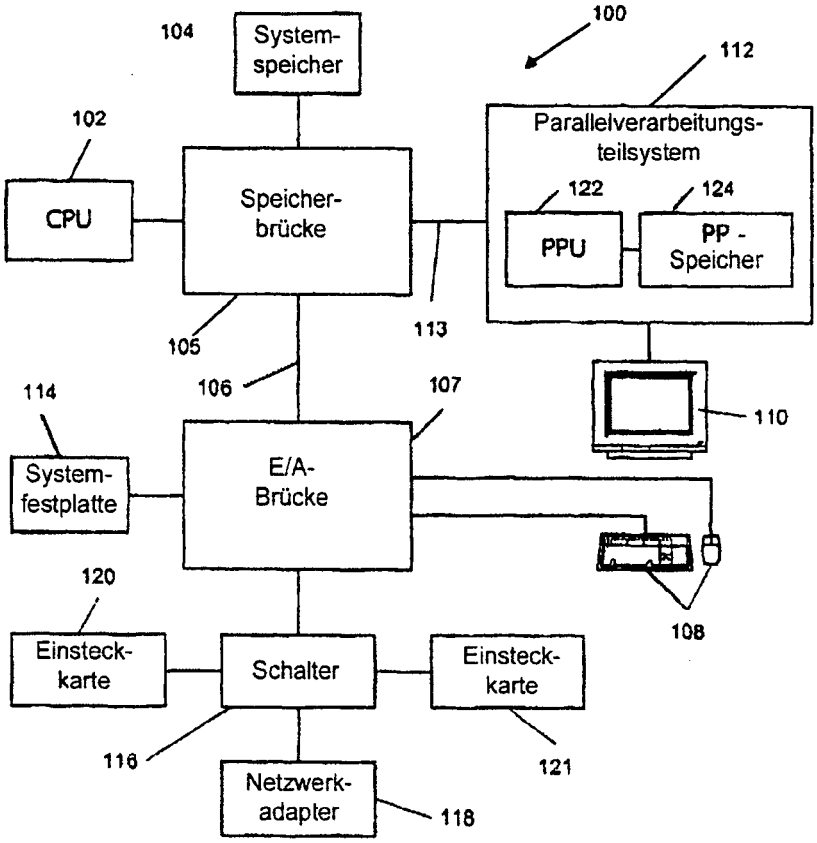


FIG. 1

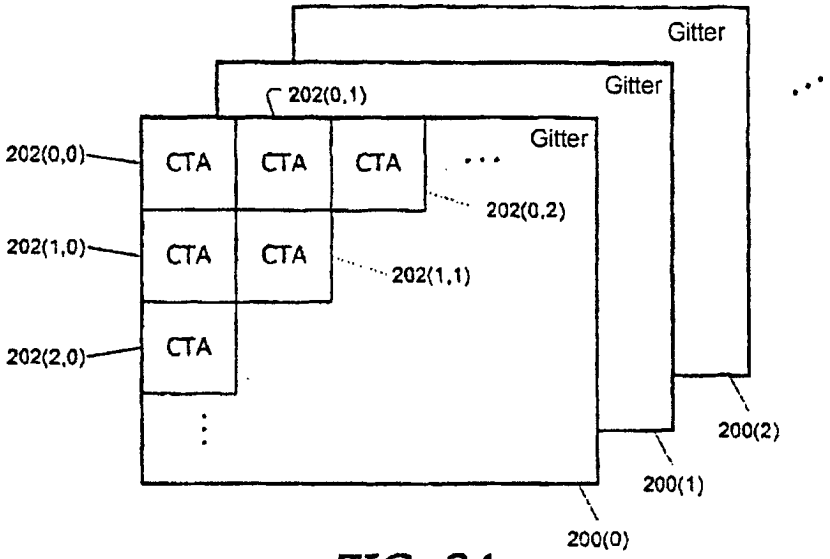

Gitter
Gitter
Gitter
202(0,1)
202(0,0)
202(1,0)
202(2,0)
CTA
CTA
CTA
CTA
CTA
CTA
202(0,2)
202(1,1)
200(2)
200(1)
200(0)

FIG. 2A

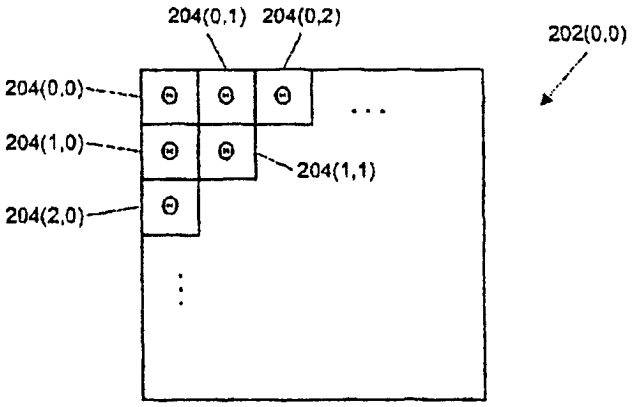

204(0,1)
204(0,2)
202(0,0)
204(0,0)
204(1,0)
204(2,0)
204(1,1)

FIG. 2B

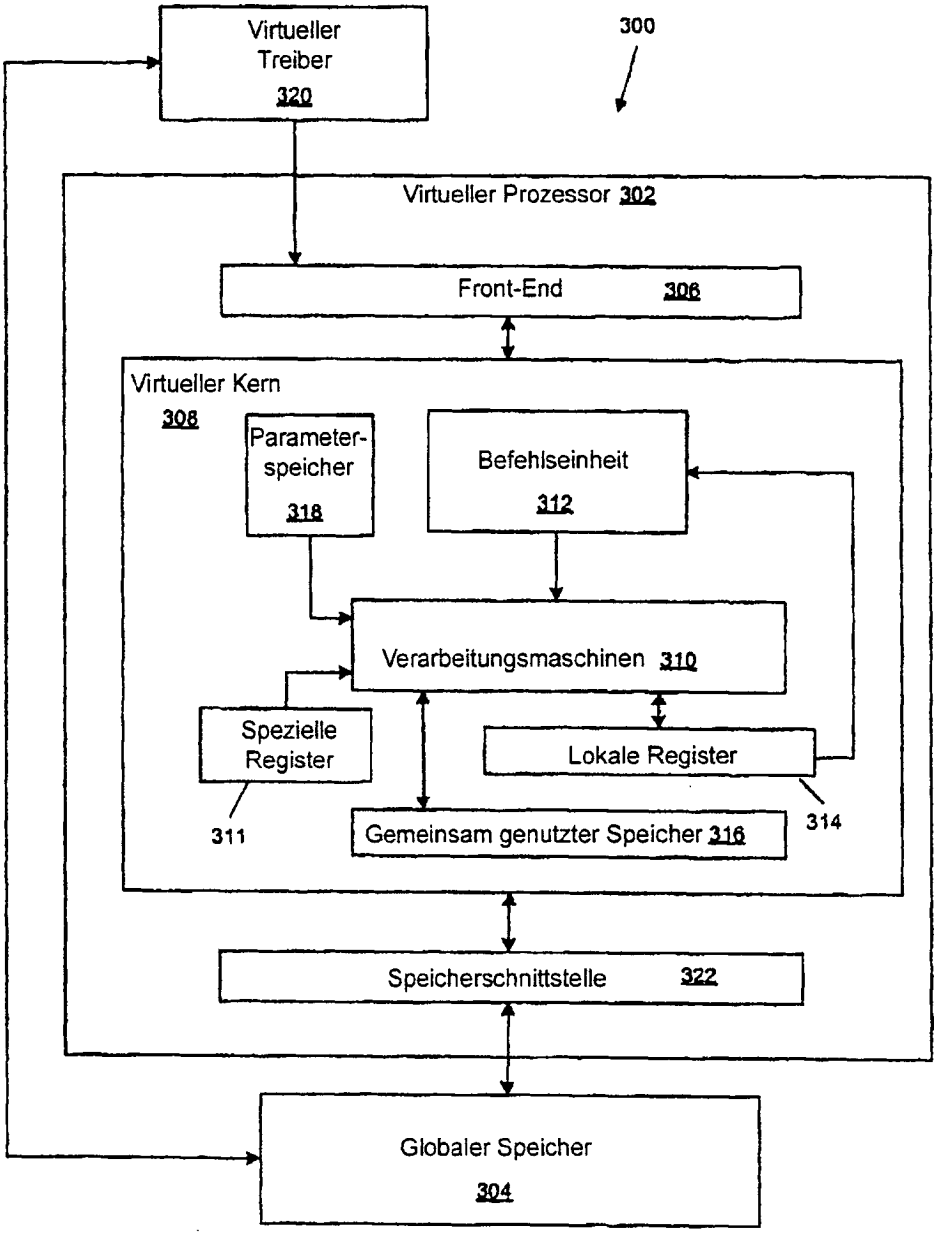

Virtueller Treiber 320
300
Virtueller Prozessor 302
Front-End 306
Virtueller Kern 308
Parameter-speicher 318
Befehlseinheit 312
Verarbeitungsmaschinen 310
Spezielle Register
Lokale Register
311
314
Gemeinsam genutzter Speicher 316
Speicherschnittstelle 322
Globaler Speicher 304

FIG. 3

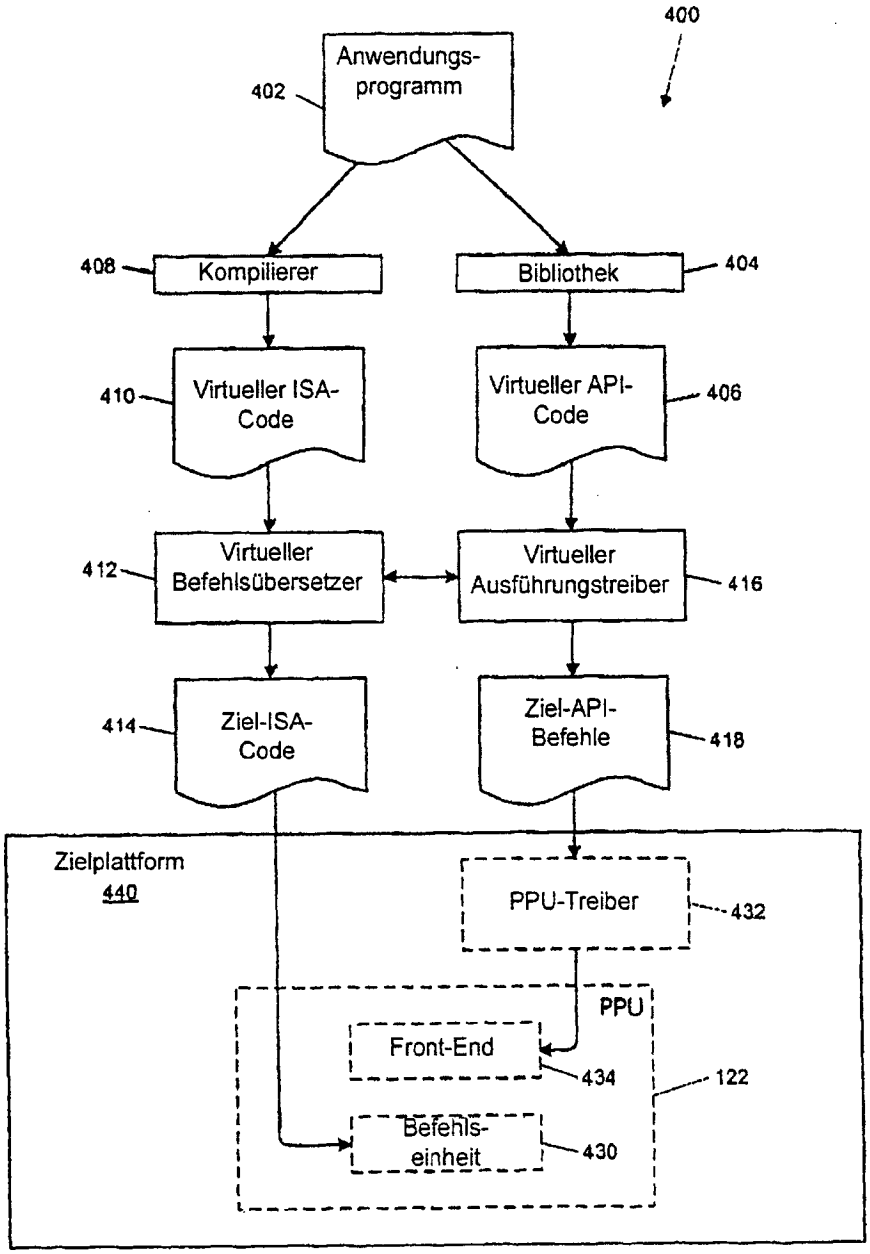

400
Anwendungs-programm
402
Kompilierer
408
Bibliothek
404
Virtueller ISA-Code
410
Virtueller API-Code
406
Virtueller Befehlsübersetzer
412
Virtueller Ausführungstreiber
416
Ziel-ISA-Code
414
Ziel-API-Befehle
418
Zielplattform
440
PPU-Treiber
432
PPU
Front-End
434
122
Befehls-einheit
430

FIG. 4

500

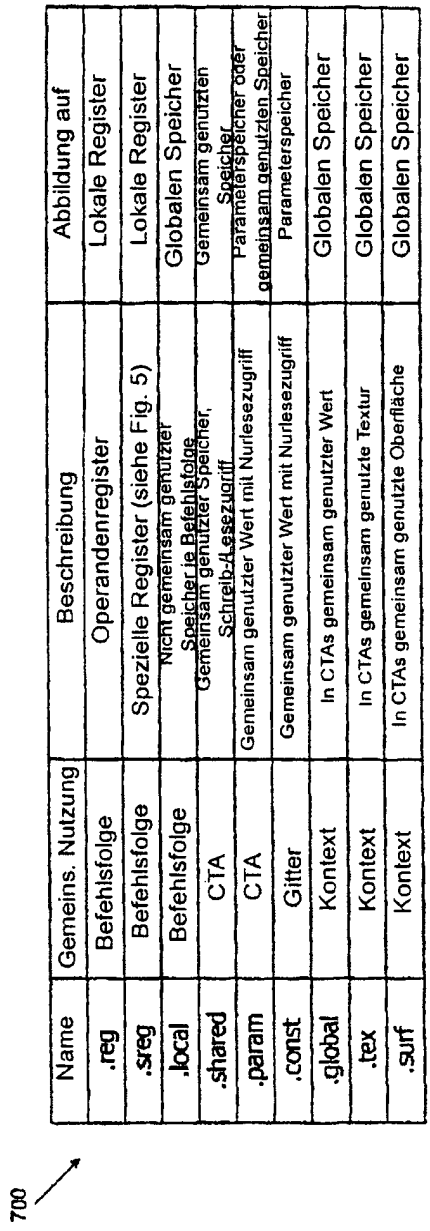
Name	Beschreibung
%ntid.x, %ntid.y, %ntid.z	CTA-Dimensionen
%tid.x, %tid.y, %tid.z	Befehlsfolge-ID innerhalb einer CTA
%nctaid.x, %nctaid.y, %nctaid.z	Gitterdimensionen
%ctaid.x, %ctaid.y, %ctaid.z	CTA-ID innerhalb eines Gitters
%gridid	Gitter-ID

FIG. 5

600

Typ	Beschreibung	Zulässige <n>
.b<n>	Nicht-typisierte Variable	1, 8, 16, 32, 64
.s<n>	Signierte ganze Zahl	8, 16, 32, 64
.u<n>	Unsignierte ganze Zahl	8, 16, 32, 64
.f<n>	Gleitkommazahl	16, 32, 64

FIG. 6



Name	Gemeins. Nutzung	Beschreibung	Abbildung auf
.reg	Befehlsfolge	Operandenregister	Lokale Register
.sreg	Befehlsfolge	Spezielle Register (siehe Fig. 5)	Lokale Register
.local	Befehlsfolge	Nicht gemeinsam genutzter Speicher je Befehlsfolge	Globalen Speicher
.shared	CTA	Gemeinsam genutzter Speicher, Schreib-/Lesezugriff	Gemeinsam genutzten Speicher
.param	CTA	Gemeinsam genutzter Wert mit Nurlesezugriff	Parameterspeicher oder gemeinsam genutzten Speicher
.const	Gitter	Gemeinsam genutzter Wert mit Nurlesezugriff	Parameterspeicher
.global	Kontext	In CTAs gemeinsam genutzter Wert	Globalen Speicher
.tex	Kontext	In CTAs gemeinsam genutzte Textur	Globalen Speicher
.surf	Kontext	In CTAs gemeinsam genutzte Oberfläche	Globalen Speicher

FIG. 7

800

Befehl	Effekt
add.<type> d, a, b	d = a+b
sub.<type> d, a, b	d = a-b
mul.<type> d, a, b	d = a*b
div.<type> d, a, b	d = a/b
mad.<type> d, a, b, c	d = [a*b] + c
fma.<type> d, a, b, c	d = a*b + c
sad.<type> d, a, b, c	d = |a-b| + c
rem.<type> d, a, b	d = a mod b
abs.<type> d, a	d = |a|
neg.<type> d, a	d = -a
min.<type> d, a, b	d = min(a,b)
max.<type> d, a, b	d = max(a,b)
frc.<type> d, a	d = a - floor(a)
sin.<type> d, a	d = sin a
cos.<type> d, a	d = cos a
atan2.<type> d, a, b	d = $\tan^{-1}$ (a/b)
lg2.<type> d, a	d = $\log_2$ a
ex2.<type> d, a	d = 2^a
rcp.<type> d, a	d = 1/a
sqrt.<type> d, a	d = sqrt(a)
rsqrt.<type> d, a	d = 1/sqrt(a)

FIG. 8A

810

Befehl	Effekt
dot.<type> d, **a**, **b**	d = sum(a[i]*b[i])
cross.<type> **d**, **a**, **b**	**d = a X b**
mag.<type> d, a	d = sqrt(sum(a[i]*a[i]))

vred.<op>.<type> d, a	d = a[0];
	für i = 1 auf Länge {
	d = <op>(d, a[i])
	}

FIG. 8B

820

Befehl	Effekt
sel.<type> d, a, b, c	d = c ? a : b
set.<cmp>.<type> d, a, b	t = a <cmp> b; d = t ? ~0 : 0
setb.<cmp>.<bop>.<type> d, a, b, c	t = a <cmp> b; d = (t <bop> c) ? ~0 : 0
setp.<cmp>.<bop>.<type> d1 | d2, a, b, c	t = a <cmp> b; d1 = (t <bop> c) d2 = (!t <bop> c)

FIG. 8C

830

Befehl	Effekt
and.<type> d, a, b	d = a & b
or.<type> d, a, b	d = a | b
xor.<type> d, a, b	d = a ^ b
not.<type> d, a	d = ~a
cnot.<type> d, a	d = !a
shl.<type> d, a, b	d = a << b
shr.<type> d, a, b	d = a >> b

FIG. 8D

840

Befehl	Effekt
cvt.<dtype>.<atype> d, a	d = (dtype) a
cvt.<mode>.<dtype>.<atype> d, a	d = (dtype) a, gerundet je <mode>

FIG. 8E

Befehl	Effekt
mov.<type> d, a	d = a
ld.<space>.<type> d, <src>	Beladen des Registers d mit Daten, die durch <src> identifiziert wurden
st.<space>.<type> <dst>, a	Speichern der Daten im Register a an dem Ort, der durch <dst> identifiziert wurde
tex d, [t, x, y]	d = Textur(t,x,y)
suld d, [s, x, y]	d = Oberfläche(s,x,y)
sust [s, x, y], a	Oberfläche(s,x,y) =a

850

FIG. 8F

Befehl	Effekt
bra <target>	gehe zu <target>
call <rv> fname <args>	Funktions-/Subroutinenruf
ret	Rückkehr von Funktion/Subroutine
exit	Abbrechen der Befehlsfolge
trap	Aufrufen der Prozessorunterbrechungsroutine
brkpt	Aussetzen der Ausführung
nop	Funktionslos (keine Operation)

860

FIG. 8G

Befehl	Effekt
bar	Warten an der Sperre auf alle anderen Befehlsfolgen der CTA
Membar <space>	Warten, bis das Schreiben in den Speicher vollendet ist
atom.<space>.<op>.<type> d, <ref>, b, c	Atomisches Lesen, Aktualisieren und Zurückschreiben in einen verfügbaren gemeinsam genutzten Speicher
vote.<op> d, a	Einstellen von d für jede Befehlsfolge in einer Gruppe auf der Basis des Umstandes, ob a in den Befehlsfolgen wahr oder falsch ist

870

FIG. 8H

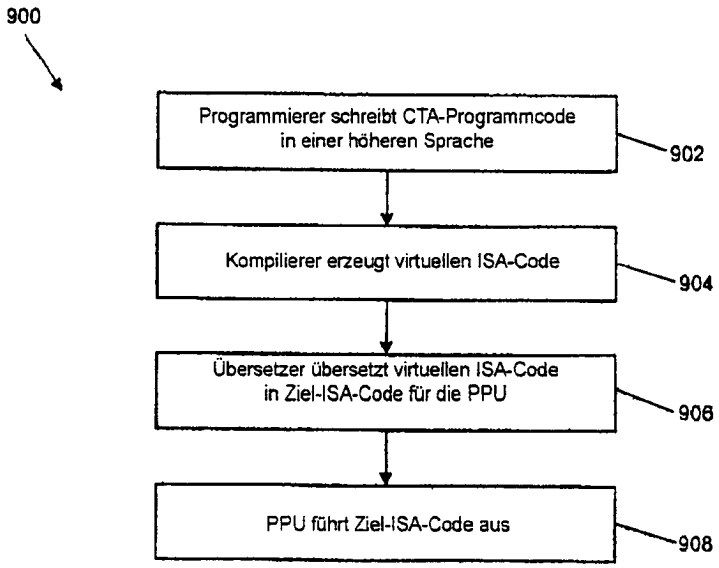


FIG. 9

1000

Funktion	Parameter
initCTA	cname, ntid.x, ntid.y, ntid.z
setCTAProgram	cname, pname
setCTAInputArray	cname, baseAddr, Größe
setCTAOutputArray	cname, baseAddr, Größe
setCTAParams	cname, np, <list>
initGrid	gname, cname, nctaid.x, nctaid.y, nctaid.z
setGridInputArray	gname, baseAddr, Größe
setGridOutputArray	gname, baseAddr, Größe
setGridParams	gname, np, <list>
launchCTA	cname
launchGrid	gname

FIG. 10