Dynamisches Resourcen Scheduling auf Grafik-Prozessoren*

Markus Steinberger
steinberger@icg.tugraz.at

Abstract: Das Ziel dieser Arbeit ist die Rechenleistung moderner Grafik Prozessoren einer breiteren Klasse von Algorithmen zur Verfügung zu stellen. Dazu wird ein neues Ausführungsmodell präsentiert, welches die effiziente Abarbeitung von Algorithmen mit inhomogenem, zeitlich veränderlichem Parallelismus ermöglicht. Ergebnis unserer Forschung ist nicht nur der zurzeit schnellste Warteschlangenalgorithmus für Grafikprozessoren, der effizienteste Speicherverwalter für massiv parallele Architekturen und der momentan einzige autonome Scheduler auf Grafik Prozessoren mit Unterstützung verschiedener Granularitäten von Parallelismus, sondern auch eine Weiterentwicklung des momentanen Standes der Technik in den Bereichen Bildsynthese, Visualisierung und geometrischer Algorithmen.

1 Einleitung

Innerhalb der letzten Jahre hat ein Paradigmenwechsel im Bereich der Computertechnologie begonnen. Da die Chipproduktion an die physikalische Grenzen gestoßen ist, ist eine Erhöhung der Prozessortaktraten nicht mehr gewinnbringend. Der einzige Ausweg, um die weiterhin steigende Nachfrage nach immer mehr Rechenleistung bedienen zu können, ist Parallelisierung. Der Beginn der parallelen Verarbeitung wurde mit der Produktion der ersten Zweikern-Prozessoren am Beginn des 21. Jahrhunderts eingeleitet. Während diese ersten Mehrkern-Prozessoren ihre Rechenleistung einfach damit erbrachten zwei Anwendungen gleichzeitig auszuführen, sind solch einfache Strategien heutzutage nicht mehr ausreichend, um die volle Rechenleistung moderner paralleler Chips voll auszureizen. Jegliche Algorithmen müssen von Grund auf darauf ausgerichtet sein, gleichzeitig von einer Vielzahl an Prozessoren abgearbeitet zu werden.

Der aktuell leistungsfähigste parallele Chip ist die Graphics Processing Unit (GPU, dt. Grafikprozessor). Sie spiegelt den Höhepunkt der Entwicklung im Bereich energieeffizienter, hoch-paralleler Rechenleistung wider. Auf einer aktuellen GPU finden sich 3000 individuelle Rechenkerne, deren effiziente Nutzung für eine Vielzahl an Forschungsgebieten essentiell geworden ist. Simulationen in diversen Gebieten wie Genetik, Molekularbiologie, Weltraumforschung, Archäologie, Neurowissenschaften oder Medizin sind Paradebeispiele für den Einsatz von Grafikprozessoren. Und obwohl die Nutzung der GPU für diese Gebiete von essentieller Wichtigkeit ist, gestaltet sich ihr Einsatz in vielen Bereichen als schwierig. Eine Portierung bestehender Software auf die GPU erreicht meist nur einen

*Englischer Titel der Dissertation: “Dynamic Resource Scheduling on Graphics Processors”

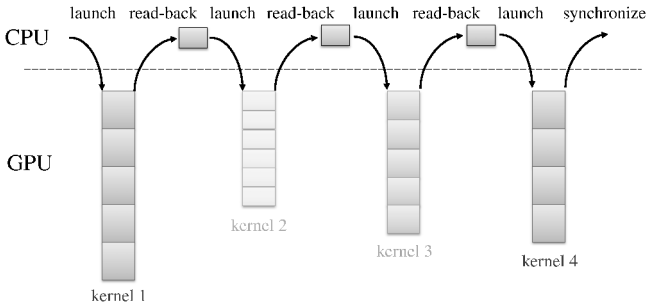


Abbildung 1: Im traditionellen Kernel-basierten Programmiermodell wird jeder Algorithmus in Kernel unterteilt. Kernel werden von der CPU aus kontrolliert, was zu einem ständigen Hin und Her zwischen CPU und GPU führt.

Bruchteil der Leistung. Die Gründe für diese Problematik finden sich im Programmier- und Ausführungsmodell von Grafikprozessoren.

Die vorliegende Arbeit beschäftigt sich mit dieser Problematik und zeigt wie neue dynamische Schedulingstrategien, welche speziell für die Ausführung auf massiv parallelen Systemen entworfen wurden, fähig sind eine unflexible GPU in eine parallele Rechenmaschine zu verwandeln, die sich an hochdynamische Algorithmen anpasst und somit die wahre Rechenleistung von Grafikchips für eine Vielzahl an Applikationen freilegt. Durch die Fortschritte, die in dieser Arbeit diskutiert werden, ist es nicht nur möglich eine neue Klasse von Algorithmen auf der GPU auszuführen, sondern auch traditionelle GPU-Algorithmen effizienter zu gestalten. Die Details der Arbeit finden sich in der Langschrift der Dissertation [Ste13], sowie in ausgewählten Publikationen in Fachzeitschriften [SKKS12, SKK+12, SKK+14a, SKK+14b].

1.1 Einschränkungen des aktuellen Ausführungsmodells

Um die Problematik hinter der Ausführung vor Algorithmen auf der GPU zu analysieren, ist es essentiell die Unterschiede zwischen der Central Processing Unit (CPU, dt. Hauptprozessor) und der Architektur eines Grafikchips zu beleuchten. Während es üblich ist Routinen zu schreiben, die jeweils einen Thread (Ausführungsstrang) auf der CPU darstellen, wird eine Prozedur auf der GPU – auch Kernel genannt – von tausenden Threads gleichzeitig ausgeführt. Während all diese Threads beginnen denselben Code auszuführen, steht es ihnen frei verschiedene Ausführungspfade zu wählen. Jedoch muss der Programmierer dabei die zusätzlichen Einschränkungen der Grafikchiphardware in Betracht ziehen. Eine GPU besteht aus einer Vielzahl an Multiprozessoren, welche Threads in kleinen Gruppen – in der Größe von z. B. 32 Threads – gemeinsam ausführen. Dieses Prinzip nennt sich Single Instruction Multiple Data (SIMD). Eine effiziennte Ausführung ist nur dann gewährleistet, wenn alle Threads innerhalb einer SIMD-Gruppe die gleichen Ausführungspfade wählen.

Expliziter Parallelismus Die erste Einschränkung des traditionellen GPU-Ausführungsmodells ist, dass ausreichend Parallelismus in jedem Abschnitt bzw. Kernel des Algorithmus zur Verfügung stehen muss. Die Anzahl an Threads, die für einen Kernel gestartet werden, muss fixiert werden bevor der Kernel ausgeführt wird. Eine dynamische Anpassung des Parallelismus ist während der Ausführung nicht möglich. Aus der Perspektive der Hardware würde es genügen, wenn eine ausreichende Anzahl an Gruppen von kohärenten Threads zur Verfügung stünde. Das Ausführungsmodell diktiert jedoch, dass tausende Threads gleichzeitig gestartet werden müssen, um eine gute Leistung zu erzielen. Dies limitiert die Anzahl an Algorithmen, die auf Grafikchips zur Ausführung gebracht werden können.

Kontrollflussänderung Die zweite Einschränkung für die Benutzung von Grafikchips für allgemeine Programmierung ergibt sich aus der Tatsache, dass jegliche Ausführung vom Hauptprozessor aus kontrolliert wird, siehe Abbildung 1. Um die Ausführung eines Kernels zu starten, muss der Hauptprozessor ein Kommando an die GPU schicken. Das Starten eines Kernels erzeugt Kosten. Zwischen zwei Kernels kann sich die Auslastung der GPU wesentlich verschlechtern. Der Datentransfer zwischen Kernels ist nur über den Hauptspeicher des Grafikchips möglich, welcher um einen Faktor 1000 langsamer ist als der lokale Speicher. In einigen Fällen ist es sogar notwendig Resultate eines Kernels von der GPU zur CPU zu übertragen. Dieser Vorgang dauert sehr lange und kann zur kompletten Inaktivität der GPU führen.

Beeinflussung der Ausführung Die dritte Einschränkung ist die Tatsache, dass es unmöglich ist, die Ausführung von Kernels zu beeinflussen, nachdem diese gestartet wurden. Der auf Grafikchips verbaute Scheduler bearbeitet Kernels nach einem einfachen first-in-first-out (FIFO) System. Daher können arbeitsintensive Hintergrundaufgaben die GPU blockieren, während hoch priorisierte Vordergrundaufgaben unvorhersehbar verzögert werden. Da die GPU keine Prioritäten unterstützt, ist es nicht möglich Aufgaben mit verschiedenen Charakteristiken gleichzeitig auszuführen.

Dynamische Speicherverwaltung Die vierte Einschränkung ist durch die Abwesenheit einer effizienten dynamischen Speicherverwaltung gegeben. Dynamische Speicherverwaltung ist ein essentieller Bestandteil jedes Betriebssystems, und praktisch jedes Computerprogramm benötigt diese Fähigkeit, um auf Eingaben reagieren zu können. Die einzige dynamische Speicherverwaltung, die es bis jetzt auf Grafikprozessoren gibt, wird von NVIDIA im CUDA Toolkit [NVI13] bereitgestellt. Der zur Verfügung gestellte Speicherverwalter ist langsam, unzuverlässig und skaliert schlecht für eine große Anzahl an Threads.

Kenntnis von Zeit Die fünfte Einschränkung ist die Tatsache, dass die GPU keine Kenntnis von Zeit zulässt. Grafikprozessoren stellen weder eine gemeinsame Zeitquelle zwischen den einzelnen Kernen noch zwischen CPU und GPU zur Verfügung. Somit ist es unmöglich einen Algorithmus an die bereits verstrichene Zeit oder die noch zur Verfügung stehende Zeit anzupassen. Die Kenntnis von Zeit ist für viele Problemstellungen, im Speziellen für Echtzeitanwendungen, welche sich an Deadlines orientieren müssen, äußerst wichtig.

1.2 Ziele der Arbeit

Diese Arbeit bietet Lösungen für alle zuvor aufgelisteten Probleme des aktuellen GPU-Schedulingmodells. Die Ziele der Arbeit lassen sich in sechs Punkten zusammenfassen:

O1 Mit Hilfe neuartiger Techniken wird eine autonome Ausführung von dynamischen Algorithmen mit Höchstleistung auf Grafikprozessoren ermöglicht.

O2 Aufgaben mit variabler Parallelität lassen sich effizient in massiv parallelen Umgebungen wie z. B. auf Grafikprozessoren ausführen.

O3 Eine dynamische Speicherverwaltung, welche Anfragen von zehntausenden Threads gleichzeitig erfüllen kann, wird ermöglicht.

O4 Das Scheduling auf Grafikprozessoren wird kontrollierbar und lässt sich an dynamisch veränderliche Bedingungen anpassen.

O5 Ein autonomes Scheduling kann für die Grafikhardware suboptimale Ausführungscharakteristika erkennen und effiziente Konfigurationen herstellen.

O6 Ein Schedulingsystem, das **O1**-**O5** erfüllt, erweitert die Bandbreite an Algorithmen, die auf massiv paralleler Hardware ausführbar sind.

Die Zielvorgaben werden dann als erfüllt angesehen, wenn sie sowohl in fundierten Teiltests als auch praktischen Anwendungen signifikante Vorteile gegenüber dem traditionellen GPU-Ausführungsmodell zeigen und die Umsetzung von Algorithmen auf der GPU erlauben, welche zuvor nicht für eine Ausführung auf Grafikprozessoren geeignet schienen.

2 Scheduling Modell

Zur Umsetzung neuartiger Schedulingstrategien auf Grafikprozessoren ist es zunächst notwendig, das vorherrschende Programmiermodell (das sogenannte Flussverarbeitungsmodell) durch ein flexibleres Modell zu ersetzen. Im Unterschied zum Flussverarbeitungsmodell geht das von uns vorgestellte Modell – Softshell – nicht davon aus, dass alle Daten vor der Ausführung im Speicher vorliegen, sondern sieht Algorithmen vielmehr als dynamische, sich anpassende, datenabhängige Ausführungspfade. Um dieses Modell umzusetzen, definieren wir fünf Basisbausteine:

- **Arbeitselemente** beschreiben Daten, welche von einem Thread bearbeitet werden.
- **Arbeitsfragmente** beschreiben Daten, welche von einer kleinen Gruppe von Threads gemeinsam bearbeitet werden.
- **Arbeitspakete** beschreiben Daten, welche von größeren Threadgruppe bearbeitet werden. Zusätzlich muss es zwischen diesen Threads Kommunikationskanäle geben.
- **Prozeduren** beschreiben die Arbeitsschritte, die für ein Arbeitselement, eine Arbeitsfragment oder ein Arbeitspaket ausgeführt werden sollen.
- **Ereignisse** werden von Benutzereingaben ausgelöst und initiieren die Ausführung von Algorithmen, indem sie Arbeitselemente, -fragmente oder -pakete generieren.

Mit Hilfe dieser fünf einfachen Bausteine ist es nicht nur möglich jeden Algorithmus des Flussverarbeitungsmodells abzubilden, sondern eine wesentlich größere Klasse an parallelen Algorithmen zu beschreiben. Durch den Einsatz von Softshell wird jede noch so kleine Ansammlung von Parallelismus darstellbar und kann an ein Schedulingsystem weitergereicht werden.

3 GPU-Schedulingalgorithmen

Zur Umsetzung aller Ziele der Arbeit ist es notwendig eine Reihe von grundlegenden Algorithmen auf massiv parallelen Architekturen umzusetzen. Die für die Arbeit wichtigsten Erneuerungen lassen sich in vier Kategorien unterteilen: Warteschlangenalgorithmen, Arbeitspaketprioritäten, dynamische Speicherverwaltung und Ausführung von heterogenen Prozeduren.

Parallele Warteschlangen Den Grundstein vieler Schedulingsysteme bilden Warteschlangenalgorithmen. Eine Warteschlange erlaubt es Arbeitselemente, -fragmente, und -pakete zu sammeln, zu kombinieren und zur Ausführung zu bringen. Bisherige Warteschlangen für parallele Architekturen sind immer davon ausgegangen, dass sogenannte *nicht blockierende* Algorithmen die beste Leistung erzielen. In dieser Arbeit zeigen wir, dass Warteschlangenalgorithmen, die in die Kategorie von *blockierenden* Algorithmen fallen, ihren *nicht blockierenden* Gegenstücken weitaus überlegen sein können. Unsere Warteschlange ist dem bisherigen Stand der Technik weitaus überlegen und kann somit als die schnellste Warteschlange für massiv parallele Architekturen angesehen werden.

Dynamische Arbeitspaketprioritäten Einer der wichtigsten Punkte des vorgestellten Schedulingsystems sind Prioritäten. Mit Hilfe von feingranularen Prioritäten auf dem Level von individuellen Arbeitspaketen ist es möglich, eine Vielzahl an Schedulingstrategien umzusetzen. Leider ist es nicht möglich Prioritätswarteschlangen basierend auf sortierten Listen oder Halden zu verwenden, da die Synchronisationskosten auf Grafikprozessoren zu hoch sind. Deshalb präsentieren wir einen adaptiven, progressiven, nicht invasiven Warteschlangensortierungsansatz, der jegliche Synchronisation mit dem Einfüge- und Entfernungsvorgängen der Warteschlangen vermeiden kann.

Dynamische Speicherverwaltung auf massiv parallelen Prozessoren Während dynamische Speicherverwaltung auf Hauptprozessoren ein gut studiertes Gebiet darstellt, lässt sich diese Forschung nicht direkt auf Grafikprozessoren übertragen, da deren Architektur zu unterschiedlich ist. Um Speicheranfragen von zigtausenden von Threads effizient bearbeiten zu können, schlagen wir einen neuen dynamischen Speicherverwalter für massiv parallele Architekturen vor: ScatterAlloc. ScatterAlloc vermeidet Kollisionen und somit Synchronisationspunkte, indem es Anfragen über eine Hashfunktion an verschiedene Stellen im Speicher verteilt. Durch eine geschickte Wahl dieser Hashfunktion ist es nicht nur möglich

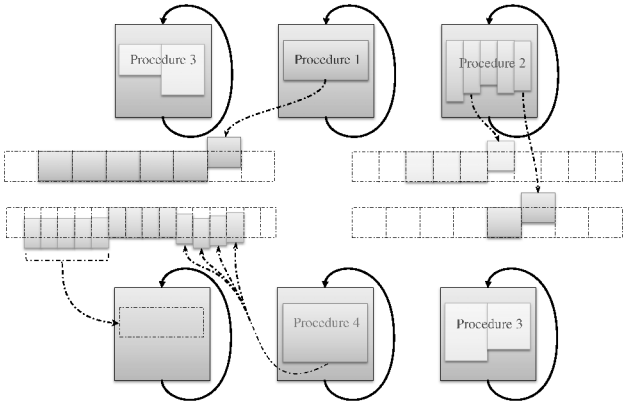


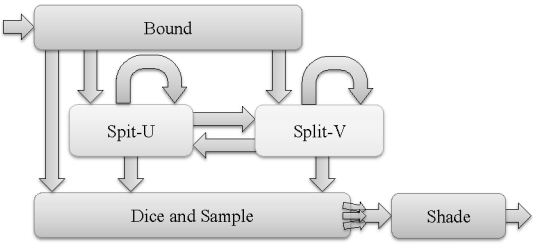
Abbildung 2: Unser Versatile Megakernel passt sich dynamisch an die ankommenden Arbeitspakete an und ermöglicht somit ein Maximum an aktiven Threads.

Anfragen zu verteilen, sondern auch Distanzen zwischen benachbarten Threads zu optimieren, um Speicherzugriffsmuster zu erzeugen, die auf Grafikhardware effizient abgearbeitet werden können. Durch dieses sorgfältige Design stellt ScatterAlloc den zurzeit effizientesten dynamischen Speicherverwalter für Grafikprozessoren dar und liegt sogar einige Größenordnungen vor den kommerziellen Implementierungen der Grafikkartenhersteller.

Ausführung von heterogenen Prozeduren Um die Rechenkraft von Grafikchips auch dynamischen Algorithmen mit verteiltem Parallelismus zur Verfügung stellen zu können, ist es essentiell, dass jegliche Teile eines Algorithmus mit höchster Rechenleistung ausgeführt werden können. Deshalb stellen wir einen Verarbeitungs- und Schedulingalgorithmus vor, welcher die zur Verfügung stehenden Ressourcen dynamisch verteilt und sich selbst an die Charakteristika der zur Ausführung eingereihten Arbeitspakte anpasst, siehe Abbildung 2. Wegen seiner Anpassungsfähigkeit nennen wir diesen Algorithmus *Versatile Megakernel*. Mit Hilfe eines einzigen Kernelstarts übernimmt der Versatile Megakernel die Kontrolle über die GPU und startet eine Maximalanzahl an Arbeiterthreads. Jede Gruppe von Arbeiterthreads lädt Arbeitseinheiten aus der globalen Warteschlange und stellt den ankommenden Arbeitseinheiten die benötigten Ressourcen zur Verfügung.

4 Anwendungsbeispiele

Die vorgestellten Schedulingalgorithmen zeigen nicht nur in synthetischen Tests hervorragende Leistungen, sondern auch erhöhte Leistungspotentiale in einer Vielzahl an Anwendungsbeispielen. Im folgenden Abschnitt zeigen wir eine Auswahl an Algorithmen, die von diesen neuen Schedulingalgorithmen profitieren bzw. sich erst mit Hilfe dieser Algorithmen auf Grafikprozessoren umsetzen lassen.



(a) Die Reyes Pipeline lässt sich mit fünf Prozeduren modellieren. Die Charakteristiken der Prozeduren unterscheiden sich stark mit 1 bis 256 aktiven Threads und 16 bis 63 benötigten Registern.

(b) Der Killeroo Datensatz besteht aus 11532 Eingabeprimitiven, die in Echtzeit in 99 Millionen Mikroflächen umgesetzt werden.

Echtzeit Reyes Pipeline Die Reyes Pipeline [CCC87] wird hauptsächlich in Filmproduktionen eingesetzt, um hochqualitative Bilder zu erzeugen. Eine Implementierung auf Grafikprozessoren gestaltet sich schwierig, da die Pipeline rekursiv, irregulär, und expandierend ist. Die grundlegende Idee hinter Reyes ist die Unterteilung der Szenengeometrie in eine Vielzahl an Mikroflächen. Für den gesamten Vorgang werden normalerweise vier bis sechs Abschnitte benötigt, welche sich, wie in Abbildung 3a dargestellt, in Softshell umsetzen lassen. Bisherige Ansätze versuchten die Pipeline in mehrere Kernel aufzuteilen [PO08, ZHR+09, TPO10]. Mit unseren Schedulingalgorithmen wird die gesamte Pipeline als eine Ausführungseinheit auf dem Grafikprozessor abgearbeitet. Für das Beispiel in Abbildung 3b werden aus 11532 Eingangsprimitiven mehr als eine Million Arbeitsbeschreibungen generiert und zur Ausführung gebracht. Temporär liegen dabei 99 Millionen Mikroflächen vor. Auf einem aktuellen Grafikprozessor (NVIDIA GTX Titan) benötigt diese Simulation dank der vorgestellten Schedulingalgorithmen nur 25ms.

Bildbasiertes Rendering mit Kamerasynchronisation Mit Hilfe des sogenannten Image-based Visual Hull (IBVH) Algorithmus lassen sich Tiefenbilder für beliebige virtuelle Kamerapositionen aus einer Reihe von segmentieren Eingabebildern erzeugen [MBR+00]. Jedoch ist der Algorithmus meist noch zu aufwendig, um eine komplette Szene in Echtzeit zu rekonstruieren. Es ist jedoch möglich, statische Teile einer Szene mit geringeren Frequenzen zu rekonstruieren und somit Rechenzeit einzusparen. Die zu rekonstruierenden Teile der Szene werden normalerweise über einen Schwellenwert definiert. Dieser Schwellenwert beeinflusst die Ausgangsqualität direkt, die Ausführungszeit jedoch nur indirekt, welche mit der Aufnahmegeschwindigkeit der Kameras synchronisiert werden muss. Mit Hilfe der vorgestellten Schedulingalgorithmen ist dies jedoch leicht realisierbar. Jedem Bereich wird anhand bestimmter Qualitätsmerkmale eine Priorität zugewiesen. Der Scheduler rekonstruiert jene Bereiche mit der höchsten Priorität vor den übrigen. Sollte der nächste Synchronisationspunkt näher rücken, lässt sich der IBVH Algorithmus durch eine einfache Wiederverwendung der älteren Daten austauschen. Somit ist es möglich, dynamisch Qualität gegen Rechenzeit zu tauschen und gleichzeitig die bestmögliche Bildqualität zu erhalten, siehe Abbildung 4.

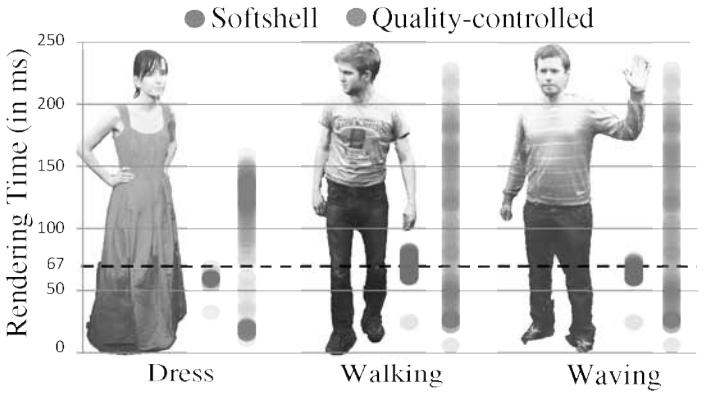


Abbildung 4: Image-based Visual Hull Rendering mit einer durch die Kameraaufzeichnungsrate vorgegebenen Ausführungszeit von $67ms$. Während mit einer qualitätsgesteuerten Anpassung des Algorithmus (rot) die Zielvorgabe (strichliert) nicht eingehalten werden kann, ist es mit dem Schedulingalgorithmus (blau) leicht möglich die Vorgaben zu erfüllen und gleichzeitig die bestmögliche Qualität zu erzielen.

Prozedurale Modellierung Mit Hilfe von prozeduraler Modellierung lassen sich komplette Welten basierend auf einem relativ einfachen Regelwerk erzeugen. Die Berechnung solcher Städte kann jedoch Stunden dauern. Da Designer einzelne Parameter der Städte in einem interaktiven Prozess anpassen müssen, führt die lange Rechendauer zu einem sehr ineffizienten Prozess. Da die für solch prozedurale Modellierung zum Einsatz kommenden Grammatiken sehr komplex sind, galt eine umfassende Abbildung dieser Regeln auf Grafikprozessoren bisher als unmöglich. Alle bisherigen Versuche beschnitten die Grammatiken stark, um Parallelität zu erzeugen und erzielten selbst für diese simplen Varianten nur geringe Vorteile im Vergleich zu CPU-Grammatiken. Mit Hilfe unserer Schedulingstrategien ist es möglich, die aktuell komplexeste Grammatik (CGA [MWH+06]) auf Grafikprozessoren abzubilden. Da die gesamte Grammatikableitung nun auf der Grafikhardware durchgeführt wird, kann die Ableitung mit der Darstellung verknüpft werden, um nur jene Teile einer Welt zu erzeugen, die auch dargestellt werden sollen. Durch diese Kombination generieren wir detaillierte Metropolen in Echtzeit, während sich ein Betrachter mit Überschallgeschwindigkeit durch die Stadt bewegen kann. Eine Beispielstadt ist in Abbildung 5 zu sehen. Mit traditionellen Techniken würde die Erzeugung dieser Stadt drei Stunden benötigen, während wir diese 20 mal innerhalb einer Sekunde neu generieren.

5 Schlussbemerkung

Das in dieser Arbeit vorgestellte GPU-Programmier- und Ausführungsmodell ist darauf ausgerichtet, verteilten und zeitveränderlichen Parallelismus nutzbar zu machen. Den Grundstein für dieses neuartige Ausführungsmodell bildet der zurzeit schnellste Warte-

Abbildung 5: Hochdetaillierte Städte mit komplexen Gebäuden können durch den Einsatz unserer Schedulingalgorithmen in Echtzeit auf einer GPU generiert werden. Die sichtbaren $28km^2$ dieser Stadt beinhalten $47\,000$ Gebäude, welche von 240 Millionen Regeln generiert werden. Mit Hilfe des vorgestellten Algorithmus generieren wir die Geometrie zur Darstellung 20 mal pro Sekunde. Somit ist es sogar möglich sich mit Überschallgeschwindigkeit durch die Stadt zu bewegen.

schlangenalgorithmus für Grafikprozessoren. Wir weisen jedem Thread eine individuelle, separat geschützte Warteschlangenposition zu, womit unser Algorithmus Anfragen in quasi konstanter Zeit beantworten kann – unabhängig von der Anzahl an Threads die gleichzeitig Zugriff suchen. Durch die Kombination mit unserem Versatilen Megakernel ist es nun möglich, Algorithmen mit diversen Ausführungscharakteristiken und unterschiedlichen Graden an Parallelismus effizient zur Ausführung zu bringen und die GPU-Auslastung auf ein Maximum zu erhöhen. Um Algorithmen die Möglichkeit zu geben sich an veränderliche Eingabeparameter anzupassen, haben wir den zurzeit schnellsten dynamischen Speicherverwalter für massiv parallele Architekturen entwickelt. Dieser Speicherverwalter reduziert die Anzahl an Zugriffsversuchen auf dieselbe Speicherstelle, indem er die Zugriffe von individuellen Threads über eine intelligente Hashfunktion verteilt. Um die Ausführung auf Grafikprozessoren beeinflussbar zu machen, führen wir feingranulare Prioritäten ein, mit deren Hilfe sich die Ausführungsreihenfolge von Arbeitspaketen sowie kompletten Algorithmen steuern lassen. Diese Prioritäten lassen zum ersten Mal Schedulingstrategien wie z. B. faires Scheduling, zeitanteilgesteuertes Scheduling, oder Earliest-Deadline-First Scheduling auf Grafikchips zu.

Da der Grad an paralleler Verarbeitung in Zukunft noch weiter ansteigen wird, ist es von essentieller Wichtigkeit genügend Parallelismus innerhalb von Algorithmen zur Verfügung zu stellen. Grundsätzlich gibt es drei Möglichkeiten zukünftige Generationen von Grafikchips mit ausreichend Parallelismus zu versorgen: Durch algorithmische Anpassungen kann ein größerer Anteil an Parallelismus freigelegt werden, mehrere Algorithmen können gleichzeitig ausgeführt werden, oder das Scheduling kann angepasst werden, sodass die Hardware die Parallelität, die in einem Algorithmus vorhanden ist, voll ausnutzen kann. In allen drei Fällen sind die in dieser Arbeit beschriebenen Ausführungs- und Programmiertechniken von großem Nutzen. Daher kann davon ausgegangen werden, dass diese Arbeit eine wichtige Rolle im Design zukünftiger GPU-Programmiersprachen, -Compiler und -Scheduler spielen wird.

Literatur

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21(4):95–102, August 1987.

[MBR+00] Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven J. Gortler, and Leonard McMillan. Image-based visual hulls. In *Proc. SIGGRAPH '00*, SIGGRAPH '00, pages 369–374. ACM, 2000.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural Modeling of Buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006.

[NVI13] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*. NVIDIA, 2013.

[PO08] Anjul Patney and John D. Owens. Real-time Reyes-style adaptive surface subdivision. *ACM Trans. Graph.*, 27(5):143:1–143:8, December 2008.

[SKK+12] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: Dynamic Scheduling on GPUs. *ACM Transactions on Graphics (TOG)*, 31(6):161, 2012.

[SKK+14a] Markus Steinberger, Michael Kenzel, Bernhard Kainz, JÃűrg MÃijller, Peter Wonka, and Dieter Schmalstieg. Parallel Generation of Architecture on the GPU. *Computer Graphics Forum*, 33(2):73–82, 2014.

[SKK+14b] Markus Steinberger, Michael Kenzel, Bernhard Kainz, Peter Wonka, and Dieter Schmalstieg. On-the-fly Generation and Rendering of Infinite Cities on the GPU. *Computer Graphics Forum*, 33(2):105–114, 2014.

[SKKS12] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, 2012.

[Ste13] Markus Steinberger. *Dynamic Resource Scheduling on Graphic Processors*. PhD thesis, Graz University of Technology, 2013.

[TPO10] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proc. High Performance Graphics*, HPG '10, pages 29–37, 2010.

[ZHR+09] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. RenderAnts: interactive Reyes rendering on GPUs. *ACM Trans. Graph.*, 28(5):155:1–155:11, December 2009.

Markus Steinberger ist Universitätsassistent in der Gruppe von Dieter Schmalstieg am Institut für Maschinelles Sehen und Darstellen an der Technischen Universität Graz in Österreich. Er hat seine Doktorarbeit in Graz im Jahr 2013 mit Auszeichnung abgeschlossen, was ihm die Ehre der Promotion unter den Auspizien des Bundespräsidenten einbrachte. Von 2013 bis 2014 arbeitete er in der Forschungsgruppe von Kari Pulli bei Nvidia in Kalifornien.

Seine Arbeiten wurden mit Best Paper Awards bei den Konferenzen NPAR'11 und Infovis'11, sowie Honorable Mention Awards bei CHI'11, Eurographics'14, sowie CHI'14 ausgezeichnet. Er ist in mehreren wissenschaftlichen Organisationen tätig, sitzt in Programmkomitees und ist Gutachter für top Journals und Konferenzen. Seine wissenschaftlichen Interessen liegen im Bereich GPU Scheduling, Scientific Visualization, Information Visualization, und Procedural Geometry.