Modellbasierte Entwicklung GPU-unterstützter Applikationen

Timo Kehrer, Stefan Berlik, Udo Kelter, Michael Ritter

Fachgruppe Praktische Informatik
Universität Siegen
{kehrer,berlik,kelter,ritter}@informatik.uni-siegen.de

Abstract: Rechenintensive numerische Anwendungen müssen oft parallel auf Multiprozessorsystemen ausgeführt werden, um akzeptable Laufzeiten zu erzielen. Ein besonders wirtschaftlicher Ansatz hierbei ist die Nutzung preisgünstiger Grafikkarten (GPU, Graphics Processing Units), die eine große Zahl von Prozessoren enthalten. Aus diversen Gründen können i.d.R. aber nicht alle Funktionen auf der GPU ausgeführt werden, d.h. eine Anwendung muß teilweise auf der CPU und GPU ausgeführt werden. Softwaretechnisch resultieren hieraus zwei Probleme: erstens sind die Programmiersprachen und Entwicklungsmethoden für GPUs deutlich verschieden von denen von Allzweckprozessoren; moderne modellbasierte Entwicklungsmethoden sind kaum verfügbar. Zweitens ist für die Kooperationen der beiden Teile einer Anwendung in erheblichem Umfang Code zu implementieren.

In diesem Papier stellen wir eine Lösung beider Probleme vor. Zunächst analysieren wir, welche Teilaufgaben bei der Implementierung von GPU-unterstützten Applikationen durch modellbasierte Entwicklungsmethoden und -Werkzeuge sinnvoll automatisiert werden können, um die Arbeitseffizienz und Korrektheit des Codes zu steigern. Sodann beschreiben wir ein auf dem Eclipse Modeling Framework basierendes Generatorsystem, das diese Implementierungsaufgaben automatisiert. Abschließend beschreiben wir an konkreten Beispielen, in welchem Umfang Code generiert werden konnte und welche Ausführungszeiten erzielt wurden.

1 Einleitung und Übersicht

Durch die stetig gestiegene Leistungsfähigkeit von Grafikkarten (GPU, Graphics Processing Units) steht inzwischen eine im Vergleich zu klassischen Rechnerclustern sehr preisgünstige Hardware zur Verfügung, die immer mehr rechenintensiven numerischen Anwendungen eine enorme Rechenleistung zur Verfügung stellen kann.

Unter dem Schlagwort “General-Purpose Computing on GPU” (GPGPU) [HA11] wurden in letzter Zeit diverse Ansätze vorgestellt, die Rechenleistung von GPUs für “allgemeine” Anwendungen außerhalb der Computergraphik verfügbar zu machen. Allgemeine Anwendungen durch eine GPU zu unterstützen ist nur unter folgenden Annahmen sinnvoll: Die Anwendung hat ohne spezielle Hardwareunterstützung schlechte Laufzeiten, und diese Laufzeiten gehen i.w. auf rechenintensive, meist numerische Teilfunktionen zurück, die gut parallelisierbar sind. Von diesen Annahmen gehen wir i.F. aus.

Die bisherigen Ansätze zur Unterstützung des GPGPU sind überwiegend sprachorientiert; sie erweitern die Programmiersprachen für GPUs in Richtung abstrakterer Konstrukte. Bisher kaum verfolgt wurde der naheliegende Gedanke, für die Entwicklung dieser Applikationen modellbasierte Technologien einzusetzen.

Ausgehend von den grundlegenden Merkmalen GPU-unterstützter, allgemeiner Applikationen (Abschnitt 2) analysieren wir in Abschnitt 3 zunächst, wie modellbasierte Technologien bei dieser Klasse von Applikationen einsetzbar sein können, wo potentiell Entwicklungsarbeit eingespart werden kann und welche Randbedingungen zu beachten sind. In Abschnitt 4 stellen wir ein Generatorsystem vor, das gemäß diesen Vorüberlegungen arbeitet. Dieses Generatorsystem wurde in mehreren Konstellationen erprobt. Die diesbezüglichen - teilweise negativen - Erfahrungen werden in Abschnitt 5 dargestellt und analysiert. Ferner wird ein daraus abgeleiteter Lösungsansatz skizziert. In Abschnitt 6 betrachten wir verwandte Lösungsansätze für die hier adressierte, softwaretechnische Problemstellung.

2 Merkmale GPU-unterstützter allgemeiner Applikationen

Allgemeine Applikationen, die man sinnvoll durch GPUs unterstützen kann, wurden schon oben grob charakterisiert. In diesem Abschnitt untersuchen wir die Randbedingungen genauer, die sich aus der Hardware und den jeweiligen Entwicklungsmethoden ergeben.

GPU-Rechenkerne haben einen sehr eingeschränkten Befehlssatz, weswegen diese Prozessoren für viele Aufgaben, z.B. Zeichenkettenverarbeitung, wenig attraktiv sind. Ferner ist von der GPU aus kein Zugriff auf Speichermedien, Kommunikationsschnittstellen usw. möglich. Aus diesen beiden und weiteren Gründen ist generell davon auszugehen, daß die Anwendungen teilweise auf der CPU laufen müssen und nur die rechenintensiven Teilfunktionen auf die GPU verlagert werden. Softwaretechnisch resultieren hieraus mehrere Probleme.

Die Daten, mit denen eine Applikationen arbeitet, stehen zunächst nur im Hauptspeicher bereit, nachdem sie z.B. von Speichermedien eingelesen oder interaktiv erfaßt worden sind. Diese Art von Daten bezeichnen wir i.F. als **primäre Daten**. Vom Hauptspeicher aus müssen die benötigten Daten auf den Speicher der GPU übertragen werden. Allerdings ist i.d.R. nur eine Teilmenge der primären Daten für die Rechenfunktionen relevant, die wir als **sekundäre Daten** bezeichnen. Nur diese Teilmenge sollte übertragen werden. Nach Beendigung der Berechnungen auf der GPU müssen Ergebnisse bzw. modifizierte Daten wieder in den Hauptspeicher zurückübertragen und in die primären Daten integriert werden. Für diese reinen Datentransfers in beide Richtungen muß in erheblichem Umfang Code entwickelt werden.

Die beiden Datentransfers kosten natürlich Zeit, die den Performance-Gewinn durch die GPU reduziert, und müssen daher optimiert werden. Da jeder einzelne Datentransfer wegen der Benutzung des (PCI-) Busses einen merklichen Grundaufwand verursacht, ist eine große Anzahl von kleinen Einzeltransfers unbedingt zu vermeiden, d.h. die Gesamtmenge der sekundären Daten muß geblockt übertragen werden. Hierzu ist eine geeignete Puffer-

verwaltung zu implementieren.

Ein weiteres Problem besteht in den Programmiersprachen und Entwicklungsmethoden für GPUs; diese sind deutlich verschieden von denen für Allzweckprozessoren [Si11]. Bei der klassischen Nutzung von GPUs zur Bildgenerierung hat die höchstmögliche Effizienz von Algorithmen absoluten Vorrang; daher sind die inzwischen verbreiteten modellgestützten Verfahren, bei denen Code generiert, also nicht von Hand optimiert wird, noch sehr unterentwickelt (s. Abschnitt 6). Letztlich werden hier Spezialsprachen genutzt; ein Beispiel ist CUDA [CUD12], das i.w. eine Erweiterung von C ist. Die CPU-seitigen und GPU-seitigen Teile der Applikationen werden daher in unterschiedlichen Sprachen programmiert, insb. wenn eine bereits existierende, modellbasiert entwickelte Applikation z.B. in Java oder C++ geschrieben ist und nachträglich durch eine GPU beschleunigt werden soll. Sofern die Typsysteme der involvierten Sprachen verschieden sind, müssen die involvierten Daten konvertiert werden, naheliegenderweise im Rahmen der Berechnung der sekundären Daten bzw. der Rückintegration der Rechenergebnisse.

Zusammengefaßt kann man eine GPU-unterstützte Applikation bzgl. der Datentransporte und Typkonversionen als ein heterogenes verteiltes System ansehen.

3 Modellbasierter Entwicklungsansatz

3.1 Analyse schematisch aufgebauter Laufzeitkomponenten

Im vorigen Abschnitt wurden grundsätzliche Randbedingungen, die sich durch die Hardwarestrukturen ergeben, geklärt und daher notwendige Funktionen zum Transport und ggf. der Konversion von Daten identifiziert. Abbildung 1 stellt die in diesem Kontext relevanten Laufzeitkomponenten dar. Für die auf dem Host (CPU) und dem Device (GPU) ablaufenden Programmteile existieren jeweils eigene Typdefinitionen. I.d.R. liegt hier ein Paradigmenwechsel von objektorientierten Sprachen seitens der CPU und der prozeduralen Programmierung auf der GPU vor. Eine Datentransportschicht übernimmt klassische Funktionen einer Middleware für heterogene verteilte Systeme, so z.B. die Verwaltung sowie das Marshalling und Unmarshalling der zwischen Host und Device zu übertragenden Objekte.

Die Implementierung der benötigten Typdefinitionen, Transport- und Konversionsfunktionen enthält i.d.R. einen hohen Anteil schematischen Programmcodes. Es liegt daher nahe, die entsprechenden Systemteile aus einer abstrakteren Spezifikation, in Abbildung 1 als Applikationsdatenmodell bezeichnet, zu generieren, worauf wir uns i.F. konzentrieren. Zustandsmodelle oder andere Modelltypen, mit denen Algorithmen spezifiziert werden können, betrachten wir hier nicht.

Wie schon erwähnt unterstellen wir, daß die CPU-seitigen Systemteile in objektorientierten Sprachen entwickelt werden. Entsprechende Codegeneratoren, welche hier ein modellbasiertes Vorgehen ermöglichen, sind für verschiedenste Modellierungsframeworks (bspw. EMF [EMF12]) und objektorientierte Zielsprachen (bspw. Java oder C++) bereits verfügbar und seit einigen Jahren erfolgreich im produktiven Einsatz. Weniger verbreitet

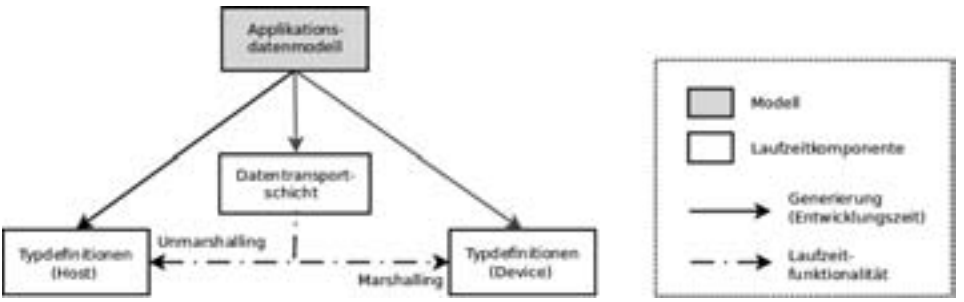

Abbildung 1: Aus Applikationsdatenmodell generierbare Laufzeitkomponenten

ist die Umsetzung objektorientierter Datenmodelle in Datenstrukturen rein prozeduraler Sprachen wie C, welche zur Programmierung auf GPUs zum Einsatz kommen. Eine komplette Lücke besteht hinsichtlich der Generierung von Transportfunktionen zwischen Host und Device. Beide Defizite werden in dieser Arbeit adressiert und sind in Abbildung 1 entsprechend grün markiert.

3.2 Entwurfsentscheidungen bei der Codegenerierung

Zur Abbildung objektorientierter Datenmodelle auf nicht-objektorientierte Sprachkonzepte existieren verschiedene Standardverfahren. Ein vorrangiges Ziel unseres Ansatzes ist die Steigerung der Entwicklungseffizienz. Objektorientierte Konzepte sollen daher auch auf der Deviceseite auf einem möglichst hohen Abstraktionsniveau umgesetzt werden, um eine komfortable Entwicklung der Kernel-Funktionen zu ermöglichen und dem Entwickler hierbei die Illusion des objektorientierten Paradigma zu vermitteln.

Entitätstypen des Datenmodells sollen daher auf Recorddefinitionen abgebildet werden. Attribute elementarer Typen werden auf Recordfelder abgebildet, ferner sollen Zugriffsmethoden zum Lesen und Setzen von Attributwerten generiert werden. Objektreferenzen sollen durch Zeiger repräsentiert werden, um so auch auf der Deviceseite eine Navigation auf “Objektnetzwerken” zu ermöglichen.

Aus dieser Entwurfsentscheidung resultieren die wesentlichen Anforderungen an die Datentransportschicht, welche ebenfalls generiert werden soll (vgl. Abbildung 1); da Haupt- und Grafikprozessor auf unterschiedlichen Speichern arbeiten, können hostseitige Zeiger nicht eins zu eins auf das Device kopiert werden. Es muss zunächst ermittelt werden, an welcher Adresse im Grafikspeicher der korrespondierende Record liegt. Das Adressmanagement sollte wie auch das notwendige Marshalling und Unmarshalling der Host-Objekte entsprechend gekapselt werden.

4 Referenzimplementierung

4.1 Zielplattform

In unserer Referenzimplementierung setzen wir C++ als objektorientierte Sprache auf dem Host ein. Deviceseitig wird Grafikkartenhardware von Nvidia eingesetzt, die *NVIDIA GeForce GTX 460*. Nvidia Grafikkarten lassen sich über die proprietäre Programmierschnittstelle *Compute Unified Device Architecture* (CUDA) [CUD12] ansprechen, welche auf der Programmiersprache C basiert. Zur Implementierung von Funktionen, welche auf der Grafikkarte gerechnet werden sollen, stellt die CUDA-API sog. Kernel bereit, welchen feingranular die verfügbaren Ressourcen der Grafikprozessoren zugeteilt werden können. Ein CUDA-Kernel arbeitet auf Daten, welche sich im Speicher der GPU befinden. Für den notwendigen Datentransfer zwischen Host und Device werden durch die CUDA-API rudimentäre Basisfunktionen zur Allokation von Speicher auf der GPU und dem Kopieren von Speicherbereichen zwischen Host und GPU (und umgekehrt) bereitgestellt.

4.2 Framework

Abbildung 2 stellt die Umsetzung der in Abschnitt 1 skizzierten Laufzeitkomponenten auf der gewählten Zielplattform dar. Grün gefärbte Funktionsblöcke interagieren mit der GPU bzw. laufen direkt auf der GPU ab. Dazu gehören GPU-seitige Datenstrukturen (in Abbildung 2 rechts unten), die darauf arbeitende Kernel-Logik (rechts oben) sowie der CUDA-Connector, welcher Datentransfers und Kernelaufrufe durchführt und koordiniert. Pfeile, welche einzelne Funktionsblöcke verbinden, stellen Funktionsaufrufe und Beziehungen dar. Schwarze Pfeile repräsentieren eine Interaktion, die lediglich hostseitig abläuft.

Zentrale Komponente des Frameworks ist der CUDA-Connector, welcher als Bindeglied zwischen Host- und Deviceseite fungiert. Hier werden die zu kopierenden Objekte verwaltet und Speicher- bzw. Pufferplatz reserviert. Sämtliche CUDA-Aufrufe sind in dieser Komponente gekapselt. Dem Programmierer werden zudem Funktionen bereitgestellt, um Datentransfers zu initiieren und einen Kernel-Aufruf anzustoßen. Ein Kernel-Aufruf wird durch den CUDA-Connector transparent um technisch notwendige Parameter, so z.B. Angaben zur Speicheradressumsetzung zwischen Host und Device, angereichert. Aus Sicht der Host-seitigen Applikationslogik (oben links) ähnelt ein Kernel-Aufruf syntaktisch damit einem regulären Funktionsaufruf der Applikationslogik auf dem Device (oben rechts), was durch den gestrichelt angedeuteten Funktionsaufruf “kernel call” dargestellt wird.

Während der CUDA-Connector in erster Linie für die Speicherverwaltung zuständig ist, wird das eigentliche Marshalling und Unmarshalling direkt an die Host-Objekte delegiert. Marshalling- und Unmarshalling-Algorithmen werden als Instanzmethoden der entsprechenden C++-Klassen, welche die Typdefinitionen auf der CPU-Seite darstellen (unten links), generiert.

Auf der Deviceseite werden die objektorientierten Typdefinitionen gemäß Abschnitt 3.2 auf C-Datenstrukturen abgebildet. Für die Entwicklung der Applikationslogik stehen ge-

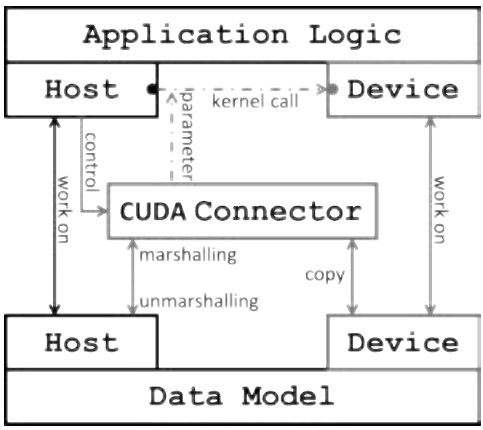

Abbildung 2: CUDA Connector als Schnittstelle zwischen Host und Device

nerierte Zugriffsfunktionen zur Verfügung, um die übertragenen "Objekte" im Speicher der GPU zu lokalisieren. Der Zugriff erfolgt hier über einen primitiven Schlüsselwert, welcher die Position in der Reihenfolge repräsentiert, in welcher die übertragenen Objekte in den CUDA-Connector geschrieben wurden. Ferner werden Funktionen zum Lesen und Setzen der Datenstruktur-Felder generiert; die Navigation über "Objektnetzwerke" geschieht zeigerbasiert.

4.3 Entwicklungsumgebung

Die realisierte Entwicklungsumgebung ist vollständig in die Eclipse IDE eingebettet. Datenmodelle der zu entwickelnden GPU-Applikationen werden mittels EMF-Ecore spezifiziert.

Sowohl der hostseitige als auch der deviceseitige Quellcode wird mittels Java Emitter Templates (JET) generiert.

Zur Erstellung eines CUDA-Projekts in der Eclipse IDE wird ein Projekt-Wizard bereitgestellt. Neben der Auswahl des Ecore-Modells sind lokale CUDA-Pfade und ein Projektname anzugeben (s. Abbildung 3). Wurde der Wizard erfolgreich ausgeführt, so findet man im Eclipse Workspace ein generiertes CUDA-Projekt vor. Generierte Programmkomponenten (host- und deviceseitige Typdefinitionen sowie der CUDA-Connector als Implementierung der Datentransportschicht) sind in dem erstellten Projekt bereits vorhanden. Ebenso Fragmente, welche die Entwicklung noch komfortabler gestalten, so z.B. ein Grundgerüst für die Main-Funktion und Implementierungsrahmen für Kernel-Funktionen. Der Entwickler kann sich somit ausschließlich auf die Implementierung der Applikationslogik konzentrieren.

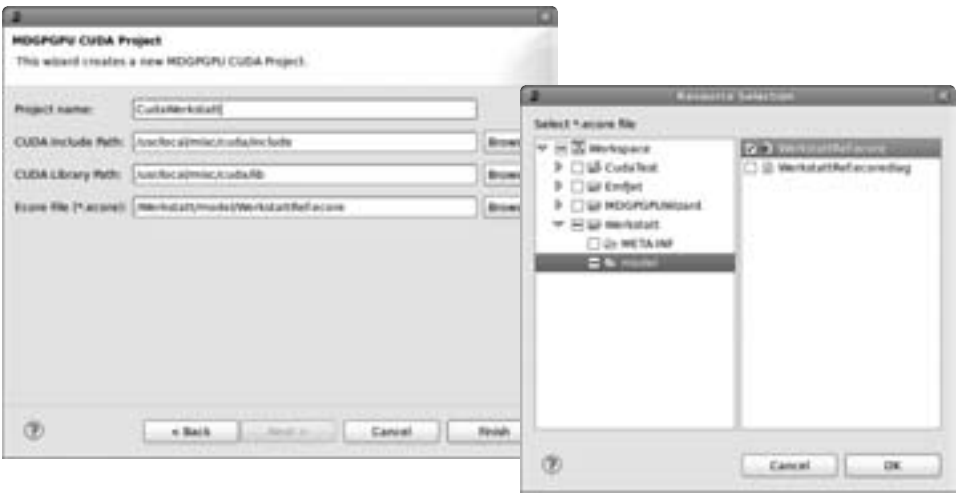

Abbildung 3: Eclipse Projekt-Wizard zur Erstellung eines CUDA-Projekts

4.4 Anschauungsbeispiel

Das folgende Anschauungsbeispiel soll zeigen, welche Artefakte auf Basis eines Applikationsdatenmodells generiert werden und wie der Entwickler diese zur Lösung seines Problems nutzen kann. Abbildung 4 zeigt das verwendete Datenmodell zur Lösung des *Traveling Salesman Problems*.

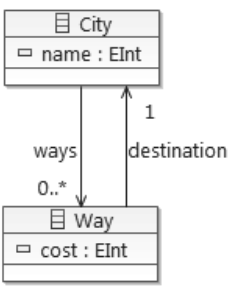

Abbildung 4: TSP Datenmodell

Generierte Programmfragmente

Eine Auswahl der generierten Artefakte ist in den Abbildungen 5, 6 und 7 zu sehen. Zur Darstellung nutzen wir UML-Klassendiagramme. Logische Gruppierungen einzelner Artefakte werden durch Pakete vorgenommen. Ferner nutzen wir zu Dokumentationszwecken Stereotypen, deren Bedeutung aus dem jeweiligen Kontext hervorgeht. Aus Platzgründen sind jeweils lediglich die wichtigsten Funktionen dargestellt.

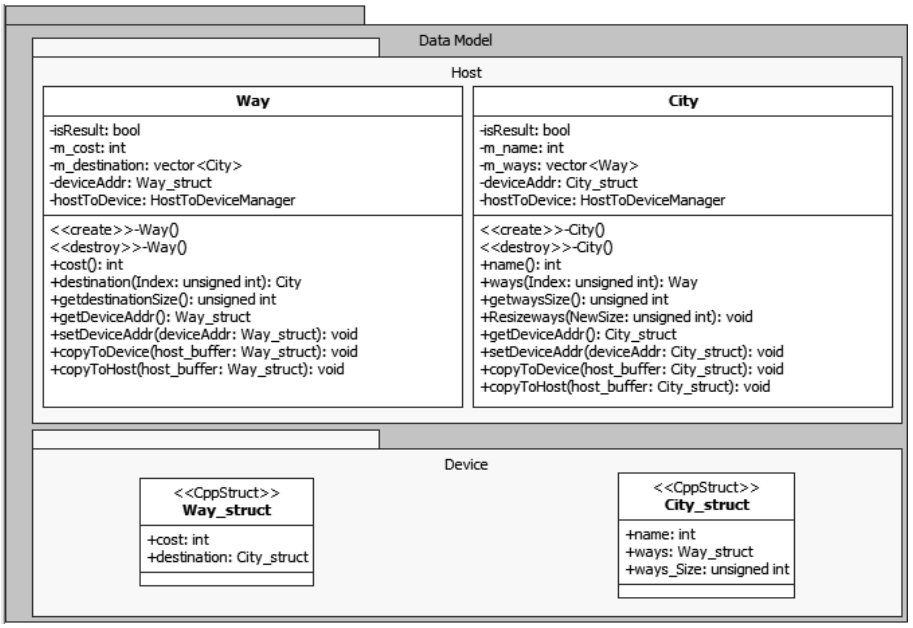

Abbildung 5: Generierte Typdefinitionen

Abbildung 5 zeigt die generierten Typdefinitionen für Host (oben) und Device (unten). Besonders relevante Methoden der hostseitigen C++-Klassen sind `copyToDevice` bzw. `copyToHost`. Diese dienen dem Marshalling und Unmarshalling. Aus Abbildung 5 nicht ersichtlich sind die konkreten Deklarationen der deviceseitigen Felder "destination" und "ways"; destination ist aufgrund der Kardinalität 1 im Applikationsdatenmodell (vgl. Abbildung 4) als einfacher Pointer (`City_struct* destination`) umgesetzt, während ways als dynamischer Array (`Way_struct** ways`) umgesetzt ist, dessen Größe durch das Feld `ways_Size` angegeben wird.

Abbildung 6 zeigt generierte Hilfsfunktionen, welche bei der Entwicklung der Applikationslogik genutzt werden können. Hostseitig wird eine Main-Funktion generiert, die bereits ein Grundgerüst für das Programm enthält. Daneben gibt es noch eine Datei namens Global, in welcher u.A. der Debug-Modus ein- bzw. ausgeschaltet werden kann. Auf der Deviceseite existieren neben einem Kernel-Grundgerüst auch "Getter" und "Setter", mit deren Hilfe auf konkrete Strukturen zugegriffen werden kann.

Abbildung 7 zeigt die wichtigsten Klassen der Datentransportschicht zwischen Host und Device. Besonders wichtig sind hier die im CUDA-Connector implementierten Stream-Operatoren. Diese legen fest, welche Objekte auf das Device zu kopieren sind. Die Erzeugung von Objekten auf der GPU wird nicht unterstützt.

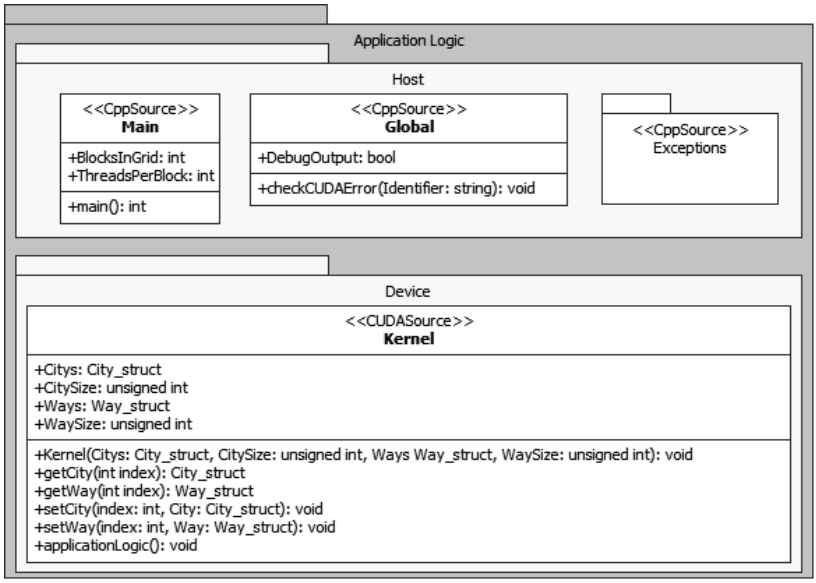

Abbildung 6: Generierte Hilfskonstrukte zur Implementierung Applikationslogik

Aufruf generierter Transportfunktionen

Der Mehrwert gegenüber dem direkten Arbeiten mit CUDA wird schnell aus folgendem Beispiel ersichtlich. Möchte man z.B. ein Array a[N], bestehend aus integer-Werten, auf das Device kopieren, so ist im Falle einer rein manuellen Implementierung folgendes Quellcodefragment nötig, welches sowohl die Speicherallokation als auch das Kopieren der eigentlichen Daten übernimmt.

```
int *dev_a;
cudaMalloc((void**)&dev_a, N * sizeof(int));
cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
```

Die generierten Framework-Komponenten gestaltet diesen Prozess deutlich einfacher. Objekte werden hier mit Hilfe des zuvor beschriebenen CUDA-Connectors und überladenen C++ Stream-Operatoren auf das Device kopiert. Aus dem obigen Quellcode wird dadurch ein intuitiver Einzeiler. Im Folgenden Codelisting wird ein City-Objekt (inklusive der referenzierten Objekte) im CUDA-Connector abgelegt.

```
Connector << &newCity;
```

Nachdem hostseitig alle Objekte angelegt, initialisiert und in den CUDA-Connector geschrieben wurden, wird der eigentliche Kernelaufuruf und der damit verbundene Kopierprozess mittels folgender Codezeile angestoßen.

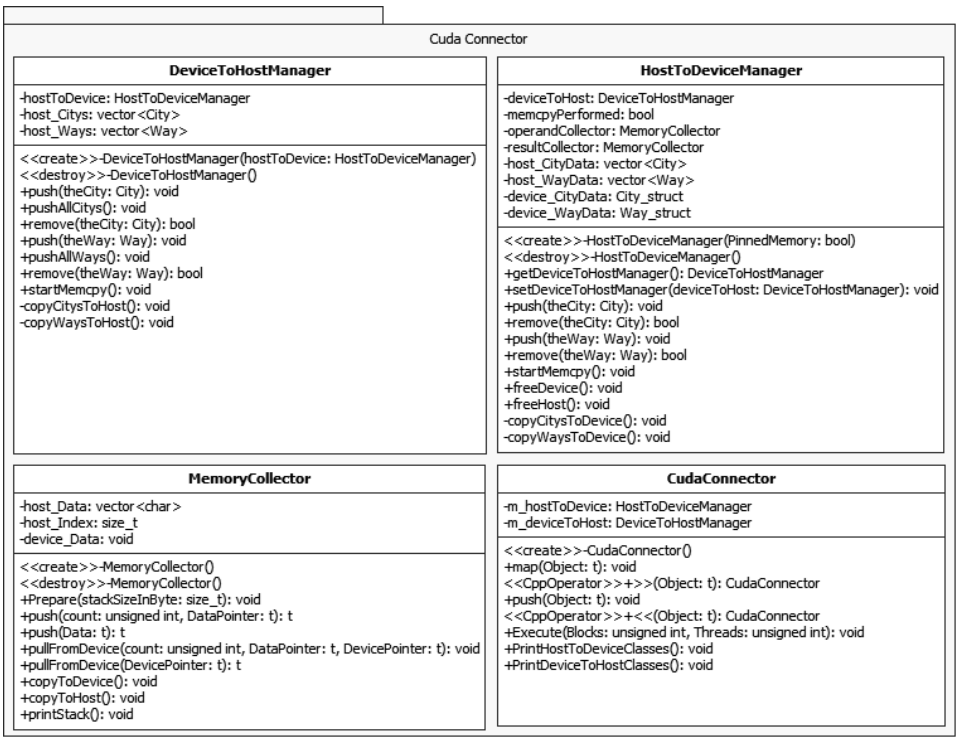

Abbildung 7: CUDA-Connector

```
Connector.Execute(BlocksInGrid, ThreadsPerBlock);
```

Hinter diesem Aufruf verbirgt sich neben der oben dargestellten Allokation von Grafikspeicher und dem Kopierprozess auch das Zurückkopieren auf den Host und die anschließende Freigabe des zuvor allokierten Speichers. Über die Werte der beiden Variablen `BlocksInGrid` und `ThreadsPerBlock`, welche hier als Parameter mit übergeben werden, wird die Anordnung der Ausführungselemente Grid, Block und Thread auf der GPU gesteuert, worauf wir hier nicht näher eingehen. Innerhalb des Kernels kann komfortabel über die in Abbildung 6 dargestellten Zugriffsfunktionen auf die sekundären Daten zugegriffen werden.

5 Evaluation

5.1 Fallstudien

Der praktische Nutzen des entwickelten Frameworks wurde anhand mehrerer Programmbeispiele untersucht. Hierzu wurden drei hochgradig parallelisierbare Anwendungen aus-

gewählt: (A) Eine einfache Vektoraddition; (B) das in Abschnitt 4.4 als Anschauungsbeispiel gewählte "Traveling Salesman Problem"; und (C), als algorithmisch komplexeres Beispiel, das in [MGR02] beschriebene Verfahren zur Propagation der Ähnlichkeit von Modellelementen. Dieses Verfahren ist u.a. im Modelldifferenzierungsframework SiDiff [SD08] implementiert und wesentliche Ursache für den hohen Rechenaufwand bei bestimmten Modelltypen und großen Modellen. Bei der Ähnlichkeitspropagation liegt zu Beginn eine Matrix vor, welche die paarweise Ähnlichkeit von Modellelementen repräsentiert. Eine weitere Matrix gibt an, welche Elementpaare benachbart, d.h. durch Beziehungen verbunden sind. In einem Berechnungsschritt werden alle Ähnlichkeiten gewichtet zu dem Nachbarn propagiert. Diese Berechnungsschritte werden mehrfach wiederholt.

Für jede der drei genannten Anwendungen wurden vier verschiedene Varianten realisiert:

1. **CPU (Array-basiert):** Die Berechnung wird rein auf der CPU durchgeführt. Die in C++ geschriebene Anwendung ist manuell implementiert und verwendet ausschließlich Arrays primitiver C++-Datentypen zur Datenhaltung.

2. **CPU (objektorientiert):** Die 2. Programmvariante läuft ebenfalls rein auf der CPU ab und ist manuell in C++ implementiert. Im Gegensatz zur Array-basierten Variante ist die Datenhaltung hier unter Ausnutzung objektorientierter Sprachkonzepte implementiert.

3. **GPU (manuell):** Die Berechnung wird bei der 3. Variante auf die GPU ausgelagert. Sämtliche Datentransfers und Speicherallokationen auf GPU-Seite werden hier manuell und ohne spezielle Werkzeugunterstützung implementiert. Als Datenstrukturen auf der GPU werden ausschließlich Arrays primitiver C-Typen verwendet.

4. **GPU (modellbasiert):** Die 4. Programmvariante wird mittels unseres modellbasierten Entwicklungsansatzes realisiert. Teile der entsprechenden Anwendungen sind somit generiert. Zur manuellen Umsetzung der Applikationslogik auf dem Device werden C-Datenstrukturen genutzt, über "Objektnetzwerke" wird ggf. über Zeiger navigiert.

5.2 Laufzeitmessungen

Im Falle der Vektoraddition wurden für alle vier Programmvarianten (vgl. Abschnitt 5.1) 10000 Additionen eines Vektors mit 1000, 10000 und 100000 Elementen durchgeführt. Die Lösung für das Traveling Salesman Problem wurde jeweils für 11 Städte und unter der Annahme vollständiger Graphen (55 Wege) berechnet. Die Ähnlichkeitspropagation wurde jeweils mit N = 1000, 4000 und 10000 durchgeführt, wobei N die Anzahl der Modellelemente repräsentiert.

Die Laufzeiten der Programme wurden mit einer CPU vom Typ Intel Core i5 650, ggf. in Kombination mit einer Grafikkarte vom Typ NVIDIA GeForce GTX 460 gemessen. Diese besitzt 336 Rechenkerne und einen Grafikspeicher von 1024 MB. Tabelle 1 zeigt

die gemessenen Ausführungszeiten der unterschiedlichen Problemstellungen für unsere vier Programmvarianten.

Programm	CPU (Arrays)	CPU (OO)	GPU (manuell)	GPU (modellb.)
Vektoraddition				
1k Elemente	0,02	0,25	0,30	0,30
10k Elemente	0,27	2,44	0,29	0,32
100k Elemente	2,83	24,48	0,33	0,46
TSP, 11 Städte	4,66	60,74	0,49	2,77
Ähnlichkeits-Propagation				
N = 3000	0,56	3,65	0,75	3,03
N = 4000	1,19	6,79	1,16	5,14
N = 10000	8,21	60,20	2,90	-

Tabelle 1: Vergleich von Programmlaufzeiten für unterschiedlichen Problemstellungen und Programmvarianten (in Sekunden)

Bei der Vektoraddition ist zu erkennen, daß eine Auslagerung der Berechnung auf die GPU erst bei einer großen Anzahl durchzuführender Additionen zu signifikanten Laufzeitverbesserungen führt. Im Falle des kleinsten Vektors, also den am wenigsten durchzuführenden Rechenoperationen, erweist sich die auf Arrays basierende CPU-Variante sogar als performanteste Implementierung. Der für den Datentransfer benötigte Zeitaufwand übersteigt hier den durch Parallelisierung auf der GPU erzielten Performanzgewinn.

Das Traveling Salesman Problem ist zur Optimierung auf der GPU gut geeignet. Wie Tabelle 1 entnommen werden kann, schneidet hier insbesondere die modellbasiert entwickelte Variante sehr gut ab. Grund hierfür ist die im Vergleich zur Anzahl der durchzuführenden Berechnungen geringe Menge an zwischen CPU- und GPU-Speicher zu übertragenden Daten; für das TSP sind lediglich 66 Objekte bzw. Strukturen nötig (11 Städte und 55 Wege).

Im Falle der Ähnlichkeitspropagation ist zu erkennen, daß objektorientierte Lösungen hier aufgrund der vergleichsweise hohen Anzahl benötigter Objekte und Pointerzugriffe gegenüber Array-basierten Lösungen deutlich inperformanter sind. Auf der CPU ist die objektorientierte Variante um den Faktor 8 langsamer als die Array-basierte; die modellbasiert entwickelte GPU-Variante gegenüber der manuellen GPU-Implementierung um den Faktor 4. Für N = 10000 übersteigt eine “objektorientierte” Berechnung auf der GPU die Grenzen des verfügbaren Speichers.

Insgesamt lässt sich festhalten, daß die Laufzeiten der modellbasiert entwickelten GPU-Varianten, mit Ausnahme der einfachen Vektoraddition, um Faktoren zwischen 4 (Ähnlichkeitspropagation mit N=3000) und 6 (TSP mit 11 Städten) schlechter sind als ihre manuell implementierten Gegenstücke. Grund dafür ist i.W. die Entwurfsentscheidung, für die generierten Typdefinitionen der Deviceseite Datenstrukturen im Gegensatz zu primitiven Arrays zu nutzen sowie Host-seitige Objektreferenzen auf der GPU-Seite durch Zeiger umzusetzen. Ist Performanz somit von höchster Priorität, so sind manuelle Implementierungen auf der GPU zu bevorzugen.

Stehen jedoch softwaretechnische Kriterien im Vordergrund und soll möglichst modellbasiert entwickelt werden, so ist dies unter geringem Mehraufwand in der Entwicklung auch für die GPU möglich, wobei für einige Problemtypen eine signifikante Steigerung der Laufzeiteffizienz erzielt werden kann: Gegenüber objektorientierten Varianten, welche rein auf der CPU ablaufen, ist für die hier beschriebenen Problemstellungen des TSP und der Ähnlichkeitspropagation eine Reduktion der Laufzeit von 60,74s auf 2,77s, von 3,65s auf 3,03s bzw. von 6,79s auf 5,14s zu beobachten.

5.3 Steigerung der Entwicklungseffizienz

Tabelle 2 stellt die Anzahl der vom Programmierer zu implementierenden Codezeilen (Lines of Code, LOC) für unsere manuell implementierten und die modellbasiert entwickelten GPU-Varianten gegenüber. Bei der Vektoraddition ist der manuelle Ansatz mit 37 Codezeilen im Vorteil gegenüber dem Generatoransatz mit 38 Codezeilen. Der Datentransfer nimmt hier nur einen sehr geringen Anteil ein. Weiterhin ist für die Einfachheit dieses Beispiels der Mehrwert einer objektorientierten Lösung der Applikationslogik nicht ersichtlich.

Komplexere Probleme wie das TSP lassen sich auf Basis des modellbasiert entwickelten Ansatzes jedoch komfortabler lösen. Im Falle des TSP sind bei der manuellen Implementierung 135 Zeilen Code zu schreiben. Die modellbasiert entwickelte Variante kommt mit 116 manuell geschriebenen Codezeilen aus, was einer Reduktion um ca. 14% entspricht. Im Falle der Ähnlichkeitspropagation beträgt die Reduktion der manuell zu implementierenden Anteile im Falle der modellbasiert entwickelten Variante ca. 31% (vgl. Tabelle 2).

Die Reduktion der manuell zu implementierenden Anteile ist hier im Falle des TSP und der Ähnlichkeitspropagation i.W. darauf zurückzuführen, daß die Typdefinitionen für primäre und sekundäre Daten ebenso wie die notwendigen Transportfunktionen generiert werden. Eine Steigerung der Entwickungseffizienz, welche auf die objektorientierte Lösung der Applikationslogik zurückzuführen ist, lässt sich auf Basis der hier gewählten, einfachen LOC-Metrik nicht messen.

Programm	GPU (manuell)	GPU (modellbasiert)
Vektoraddition	37	38
TSP, 11 Städte	135	116
Ähnlichkeits-Propagation	153	109

Tabelle 2: Manuell zu implementierende Anteile (in LOC) bei unterschiedlichen Implementierungsvarianten

5.4 Diskussion

Die Imitation der objektorientierten Strukturen zur Verwaltung der Sekundärdaten hat aus Sicht von Entwicklern den Vorteil, daß die Datenstrukturen in beiden Systemteilen sehr ähnlich sind und leicht mental aufeinander abgebildet werden können.

Nachteilig hinsichtlich der Laufzeit ist jedoch die Verwendung vieler Referenzen auf dem Device: GPU-Prozessoren und -Speicher sind nicht unbedingt optimiert für das Verfolgen von Pointerketten, ferner erfordert jede Objektreferenz im Zuge des Marshalling bzw. Unmarshalling die Umrechnung von Zeigeradressen. Nachteilig ist ferner die Speicherfragmentierung.

Die vorstehenden Nachteile können weitgehend vermieden werden, wenn primitive Typen und ggf. Arrays als Typdefinitionen für die Sekundärdaten verwendet werden. Derartige Typdefinitionen können auch sehr genau auf den Bedarf der Rechenfunktionen abgestimmt sein. Nachteilig ist hier, daß diese hoch angepassten Typdefinitionen nicht mehr via einem Standardverfahren aus den objektorientierten Datenmodellen ableitbar sind, ebenso sind die Konversionen zwischen beiden Datenstrukturen wesentlich komplizierter und u.U. nicht mehr automatisch generierbar. Gegenwärtig untersuchen wir, inwiefern sich die Ausgaben des Codegenerators durch entsprechende Steuerparameter optimieren lassen. Ggf. wird neben dem plattformunabhängigen Datenmodell, aus denen die primären Datenstrukturen generiert werden, ein verfeinertes, plattformspezifisches Datenmodell benötigt, aus welchem die sekundären Datenstrukturen generiert werden. Typischerweise sind die sekundären Datenmodelle eine Teilmenge der primären Datenmodelle.

6 Vergleich mit existierenden Arbeiten

Unter dem Schlagwort “General-purpose computing GPU” (GPGPU) sind eine Reihe von Vorschlägen publiziert worden, die Programmierung von GPUs generell bzw. für konventionelle Zwecke zu vereinfachen. So stellen bspw. Han und Abdelrahman in [HA11] die Sprache *hi*CUDA und einen zugehörigen Compiler vor, die u.a. ebenfalls das Problem adressieren, Datenstrukturen im GPU-Speicher automatisch zu generieren. *hi*CUDA erweitert CUDA um Direktiven, mit denen diese Aufgaben automatisiert werden können. *hi*CUDA ist aber nicht modellbasiert und behandelt nur die Entwicklung von Software auf der GPU, nicht die integrierte Entwicklung der Applikationsanteile auf der CPU und GPU.

Im Rahmen des Microsoft Accelerator Forschungsprojekts [TPO06, Mi11] wurde eine Bibliothek entwickelt, welche Operationen zur datenparallelen Verarbeitung von Arrays bereitstellt. Der Funktionsumfang der Bibliothek stellt eine domänenspezifische Sprache dar, welche in gängige Hochsprachen wie bspw. C++, C#, F# und Haskell eingebettet werden kann. Die zu parallelisierenden Operationen werden on-the-fly übersetzt, Implementierungen existieren für verschiedene Multikern-CPU-, FGPA-, und GPU-Hardwarearchitekturen.

Breitbart stellt mit dem CuPP-Framework [Br09] einen ebenfalls sprachbasierten Ansatz vor, der das Problem der Typkonversionen zwischen CPU und GPU adressiert. Das CuPP-Framework bietet Transformationen an, die eine in C++ gültige Klasse vom Host in eine

CUDA-konforme Repräsentation auf dem Device umwandeln. Fest in CuPP integriert ist bisher jedoch nur eine Vektor-Klasse, die genannte Transformationsmechanismen nutzt. Ist man auf zusätzliche Klassen angewiesen, so müssen diese manuell mit Hilfe der bereitgestellten Funktionen angelegt werden. CuPP ist daher weniger flexibel als unser Ansatz hinsichtlich der Sprachen, in der die Host-Programme laufen, und der Vielfalt der unterstützten Datentypen.

Auch Yan et al. widmen sich der Vereinfachung des Datentransfers zwischen CPU- und GPU-Speicher. Vorgestellt wird mit JCUDA [YGS09] ein Ansatz, der neben vereinfachtem Datentransfer auch eine Java-Schnittstelle für CUDA anbietet. Letztere wird mit Hilfe des Java Native Interface (JNI) realisiert. Unterstützte Datentypen sind neben skalaren Attributen auch Arrays. Objektorientiertes Arbeiten ist bei JCUDA nicht möglich. Datentransfers werden über die Parameter IN, OUT und INOUT näher spezifiziert. Entsprechender Code zum Kopieren der definierten Datenstrukturen wird vom Compiler automatisch erzeugt. Dieser nimmt eine Text-zu-Text Übersetzung vor, bei der aus einem JCUDA-Programm ein Java-Programm erzeugt wird, welches mittels JNI C-Code verarbeiten kann.

Da auf Seiten von CPU und GPU auf "den gleichen" Daten gearbeitet wird, liegt der Gedanke nahe, die Daten vollautomatisch zu synchronisieren; Gelado et al. [GCN+10] stellen mit Asymmetric Distributed Shared Memory (ADSM) ein datenzentriertes Programmiermodell vor, das sowohl CPU- als auch GPU-seitig auf demselben Adressraum arbeitet. Die Asymmetrie besteht darin, daß vom CPU-Speicher auf den GPU-Speicher zugegriffen werden kann, nicht aber umgekehrt. So wird zum Beispiel eine Funktion zum Allokieren von Speicher bereitgestellt, die sowohl im CPU- als auch GPU-Speicher eine Allokation vornimmt und diesen Speicherbereich auf dieselbe logische Adresse legt. Aus Sicht des Entwicklers wird dadurch auf denselben Daten gearbeitet. Dieser Ansatz weist eine gewisse Analogie zu unserer Arbeit auf. Wir arbeiten hostseitig mit Klassen und deviceseitig mit Strukturen, die durch das Framework automatisch konsistent gehalten werden.

7 Zusammenfassung und Ausblick

Die Auslagerung rechenintensiver Applikationsteile auf die GPU erfordert den Transport von Daten des CPU-Hauptspeichers in den GPU-Speicher sowie eine entsprechende Rückintegration nach Beendigung der Berechnung auf der GPU. Aufgrund der meist unterschiedlichen Sprachen und Typisierungskonzepte seitens der CPU und der GPU lassen sich GPU-unterstützte Applikation bzgl. der Datentransporte und Typkonversionen als ein heterogenes verteiltes System ansehen. Existierende Lösungsansätze zur Bereitstellung softwaretechnischer Methoden, welche eine effiziente Entwicklung GPU-unterstützter Applikationen auf möglichst hohem Abstraktionsniveau ermöglichen, sind überwiegend sprachorientiert; sie erweitern die Programmiersprachen für GPUs in Richtung abstrakterer Konstrukte.

In diesem Papier wurde ein modellbasierter Lösungsansatz vorgestellt. Kern des Ansatzes ist es, die benötigten Typdefinitionen sowie Transport- und Konversionsfunktionen auf Basis *eines* Applikationsdatenmodells zu generieren. Die Entwicklungseffizienz kann

somit gegenüber einer rein manuellen Implementierung auf der GPU für komplexe Problemstellungen deutlich gesteigert werden. Auch die Optimierung der Laufzeitperformanz gegenüber rein CPU-basierten Lösungen kann für rechenintensive, parallelisierbare Applikationen nachgewiesen werden.

Eine wesentliche Erkenntnis ist allerdings, daß die üblichen Methoden, wie aus (Daten-) Modellen Code generiert wird, bei dieser Klasse von Applikationen u.U. nicht sinnvoll ist und typischerweise zu so hohen Leistungsverlusten führt, so daß der GPU-Einsatz im Extremfall sogar sinnlos wird. Stattdessen muß die Codegenerierung an die Optimierungsbedürfnisse der jeweiligen Anwendung angepaßt werden. Hierzu schlagen wir veränderte Codegeneratoren vor, die durch entsprechenden Generator-Anweisungen konfiguriert werden.

Literatur

[Br09] Breitbart, Jens: CuPP - A framework for easy CUDA integration; p.1-8 in: Proc. IEEE Intl. Symp. Parallel and Distributed Processing; 2009

[CUD12] CUDA: Compute Unified Device Architecture, http://www.nvidia.de/object/cuda_home_new_de (Letzter Abruf: 23. Januar 2012)

[EMF12] EMF: Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/ (Letzter Abruf: 23. Januar 2012)

[GCN+10] Gelado, I.; Cabezas, J.; Navarro, N.; Stone, J.E.; Patel, S.; Wen-mei, W.H.: An Asymmetric Distributed Shared Memory Model for Heterogenous Parallel Systems; Architectural Support for Programming Languages and Operating Systems, p.347-358; 2010

[HA11] Han, T.D.; Abdelrahman, T.S.: hiCUDA: High-Level GPGPU Programming; IEEE Transactions on Parallel and Distributed Systems 22:1, p.78-90; 2011

[MGR02] Melnik, S.; Garcia-Molina, Hector; Rahm, E.: Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching; p.117-128 in: Proc. 18th Intl. Conf. Data Engineering; IEEE Computer Society; 2002

[Mi11] Microsoft Accelerator Projekt, http://research.microsoft.com/en-us/projects/accelerator/ (Letzter Abruf: 23. Januar 2012)

[SD08] SiDiff, http://pi.informatik.uni-siegen.de/sidiff/ (Letzter Abruf: 23. Januar 2012)

[Si11] Singh, Satnam: Computing Without Processors; Communications of the ACM; August 2011

[TPO06] Tarditi, David; Puri, Sidd; Oglesby, Jose: Accelerator: Using data parallelism to program GPUs for general-purpose uses; p.325-335 in: Proc. 12th Intl. Conference on Architectural Support for Programming Languages and Operating Systems; 2006

[YGS09] Yan, Yonghong; Grossman, Max; Sarkar, Vivek: JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA; European Conference on Parallel Processing, p.887-899; 2009