BACHELORARBEIT

Compiling F# Functions for GPUs

Zur Erlangung des akademischen Grades
Bachelor of Science
(BSc.)

ausgeführt am
Institut für Computersprachen
Arbeitsgruppe Programmiersprachen und Übersetzer

der Technischen Universität Wien

unter der Anleitung von
Univ.-Prof. Dipl.-Ing. Dr. Andreas Krall

durch:
Harald Steinlechner
Hofmannsthalgasse 24/14/22, 1030 Wien

Wien, August 2011

Kurzfassung

Die rasante Weiterentwicklung von Hard- und Software im Bereich der Grafikprozessoren führte zu verschiedenen Ansätzen, die hohe Rechenleistung dieser Chips auch für andere Zwecke als Realtime-Rendering zu verwenden. Leider unterscheidet sich die Architektur der GPUs (Graphics Processing Units) grundlegend von weit verbreiteten Desktop-Prozessoren, was die Verwendung erschwert.
Programmierung dieser Architekturen ist zur Zeit mittels spezieller Sprachen möglich - aus Software-Engineering-Sicht erweist sich dies häufig als problematisch. Zum einen durch die hohe Komplexität paralleler Algorithmen, zum anderen durch den Mangel ausgereifter Entwicklungsumgebungen für eben diese Sprachen. Außerdem ist die Wiederverwendbarkeit gegenüber bestehendem und zukünftigen Code sehr eingeschränkt, da sich die APIs ständig ändern. Dies sind Gründe dafür, dass das Potential der Hardware selten ausgeschöpft wird. Funktionale Programmierung bietet hohe Wiederverwendbarkeit, Abstraktion und motiviert Parallelisierung. Diese Arbeit beschäftigt sich mit der Übersetzung funktionaler Sprachen auf imperative Sprachen, welche von GPUs ausgeführt werden können um die Vorteile funktionaler Hochsprachen mit der hohen Leistung von GPUs zu vereinen.

Schlüsselwörter

source to source compilation, parallel programming, functional programming, compilers.

Abstract

Rapid improvement in programmability and hardware flexibility unleashed the power of Graphics Processing Units (GPUs) to a broad field of applications exploiting parallel program execution. Parallel programming, inherently a difficult problem in computer science coupled with a vast number of architectures, languages and lack of classical software development tools causes troubles to software engineers - eventually withholding developers from actually using GPUs effectively. Enhancing GPUs programmability or even exploiting its performance implicitly by automatic code generation is a very active research topic currently. Special purpose languages often suffer flexible development toolchains, parallelization libraries often lack functionality and flexibility. Also from and economoic point of view code reusage and stability plays a crucial role in software development. Functional programming offers composability, abstraction and is well suited for parallel programming. We present an efficient F# compiler targeting GPU programs like shaders - nevertheless our approach is general and also supports other imperative languages as well. Our work motivates a large set of applications, libraries and programming models supporting developers to unleash the power of GPUs.

Keywords

source to source compilation, parallel programming, functional programming, compilers.

Inhaltsverzeichnis

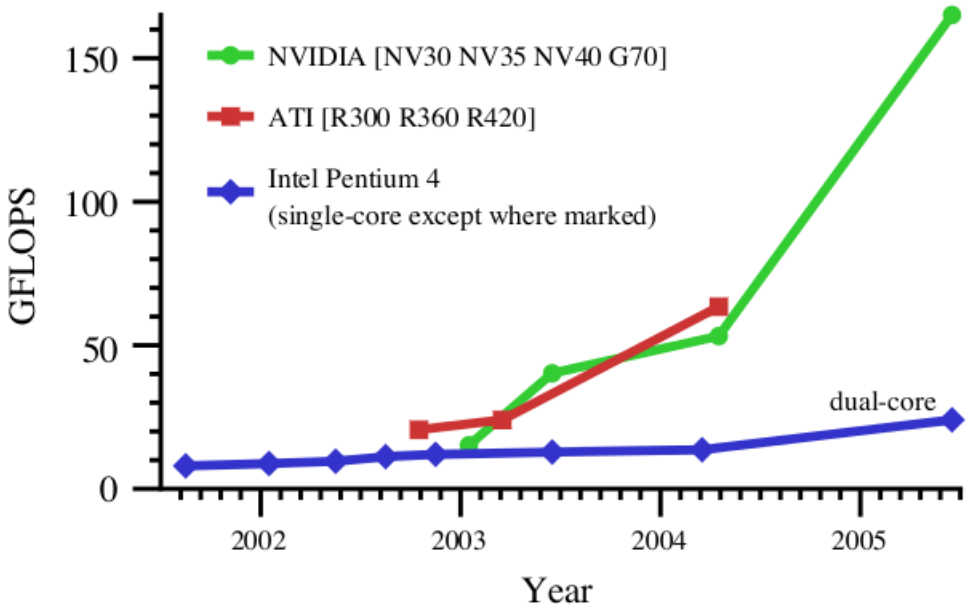


Abbildung 1: Programmierbare Floating-point Performance von GPUs (gemessen wurden multiply-add instructions als 2 floating-point operations). Figure courtesy Ian Buck, Stanford University Owens et al. [2007] .

1 Einleitung

Nach algorithmischer Optimierung von Programmen werden diese um doch noch die Performance-Anforderungen zu erfüllen häufig parallelisiert. Von linearer Beschleunigung proportional zur Anzahl der parallelen Einheiten ist man allerdings in der Praxis weit entfernt - falls die Algorithmen überhaupt parallelisierbar sind. Wie Forschungsarbeiten im Bereich paralleler Algorithmen gezeigt haben, lassen sich auf den ersten Blick sequentielle Algorithmen allerdings erstaunlicherweise recht oft parallel implementieren oder durch algorithmischer Umstrukturierung mittels paralleler Algorithmen formulieren. Jedenfalls ist diese Aufgabe äußerst schwierig und zeitaufwändig - will man die Programme nun auch noch auf speziellen parallelen Recheneinheiten ausführen, steigen die Entwicklungskosten ins unrentable.

Im Bereich der Echtzeitgrafik haben sich GPUs (Graphics Processing Units) als spezielle Hardware durchgesetzt, um die benötigten Algorithmen besonders effizient ausführen zu können. Traditionell stellt die Hardware eine Pipeline zur Verfügung, welche Rendering mittels Rasterisierung bzw. Scan-line Rendering realisiert. Getrieben durch höhere grafische Anforderungen wurde diese ursprünglich in Hardware implementierte Pipeline zusehends flexibler. Mittlerweile lässt sich diese Hardware bis zu einem gewissen Grad programmieren. Beispielsweise lässt sich der Vertex-Prozessor, welcher unter anderem für die Transformation und Projektion der 3D-Punkte zuständig ist frei programmieren um Spezialeffekte wie Animation direkt als Schritt der Pipeline implementieren zu können. Das hohe Maß

an Parallelität wirkt sich im direkten Vergleich von GPUs und CPUs stark auf die reine Rechenleistung aus. Während die Erhöhung der Taktfrequenzen von CPUs stagniert, skaliert Parallelisierung weiterhin. In Abbildung 1 wird die Entwicklung der Floating-Point-Performance von GPUs im Vergleich zu CPUs veranschaulicht.

Da derartige GPUs in Consumer Rechnern in der Regel vorhanden sind, ist die Ausnützung von GPUs für allgemeine Berechnungen erstrebenswert. Vor allem in der Forschung wird von GPUs für allgemeine Berechnungen Gebrauch gemacht - im Mainstream-Bereich ist diese Vorgehensweise *(noch)* nicht verbreitet. Mögliche Gründe dafür sind:

- Bedingt durch das ursprüngliche Designziel Computergrafiker zu unterstützen, sind APIs meist nicht flexibel genug um Programme effizient für GPUs entwickeln zu können. Die Entwicklung von CUDA NVIDIA Corporation [2010] oder auch OpenCL Chronos Group [2010] hat diesen Grund schon weitgehend entschärft.
- GPUs bieten zwar Infrastruktur für die Implementierung paralleler Algorithmen, die Programmierung gestaltet sich allerdings sehr hardwarenah und erfordert dadurch viel Wissen über den internen Aufbau der Hardware.
- Architektur-bedingt ist die Programmierung nur mittels spezieller Sprachen möglich. Dies erfordert wiederum Einarbeitungsaufwand für diese Sprachen, des weiteren muss bestehender Code portiert werden. Zusätzlich können teuer optimierte Programme durch Hardware-Weiterentwicklungen ineffizient oder sogar unbrauchbar werden.
- Es existieren zwar Entwicklungsumgebungen (wie etwa NVIDIA nSight), diese sind allerdings proprietär und im Vergleich zu etablierten IDE's unausgereifter. Debugging ist dabei besonders eingeschränkt.

Gerade aus diesen Gründen ist dieses Forschungsgebiet derzeit sehr aktiv (siehe 2). Stark vereinfacht sind folgende Strömungen zu erkennen:

- Entwicklung spezieller Sprachen, um parallele Programmiermodelle abbilden zu können.
- Parallelisierung über API's, welche Bauelemente für parallele Algorithmen bieten.

Diese Arbeit beschäftigt sich mit der Übersetzung von Programmteilen funktionaler Sprachen auf Programme, welche auf der GPU ausführbar sind. Konkret wird im vorgestellten Compiler F# als Quellsprache verwendet. Die Zielsprache ist dabei GLSL, eine Shadersprache - die Arbeit ist allerdings allgemein genug gehalten, um hier eine andere imperative High-Level Shader-Sprachen zu wählen oder auch Code für beispielsweise CUDA zu erzeugen.

Das Einsatzgebiet des vorgestellten Übersetzers ist breit: Einerseits können im API Ansatz, die notwendigen Bausteine automatisch erzeugt werden, andererseits können für Parallelisierung spezielle Sprachen mittels Embedded Domain Specific Language direkt in F# programmiert werden, wobei das optimierte GPU-Programm mittels source-to-source Übersetzung erzeugt wird. Die Arbeit ist wie folgt gegliedert:

- Das Problem der Übersetzung einer funktionalen Sprache in eine imperative Sprache wird in 2 behandelt. Die Übersetzung von F# in eine imperative und derart eingeschränkte Zielsprachesprache ist erheblich komplexer als reine *source-to-source*-Übersetzung. Eine besondere Herausforderung dieser Arbeit ist es, eine funktionale Sprache auf der GPU auszuführen, da dort häufig verwendete Werkzeuge für die Implementierung fehlen.
- Eine Beschreibung des Übersetzers, sowie die Eingangs- und Ausgangssprache ist in 3 zu finden.
- Die notwendigen Code-Transformationen sowie die Syntax und Semantik der Zwischensprachen werden in 3 behandelt.
- Als konkreter Anwendungsfall wird die Implementierung von Shadern im Bereich Realtime-Rendering demonstriert. (4).

2 Related Work

2.1 General Purpose Computation on Graphics Processing Units (GPGPU)

Grafikprozessoren (GPUs) bieten durch zahlreiche Kerne ein hohes Potenzial an Parallelität und werden historisch im Bereich der Echtzeitgrafik eingesetzt. Bedingt durch flexiblere neuartige Architekturen werden GPUs aber auch für andere algorithmisch komplexe Aufgaben verwendet. Eine detaillierte Analyse ist in [Owens et al., 2007] zu finden.
Grafikprozessoren sind meist mit mehreren hundert Kernen ausgestattet, welche das gleiche Programm parallel auf verschiedenen Bereichen der Eingabedaten ausführen. Diese Art von Parallelität wird Data-Parallelism genannt und wurde im Bezug auf GPUs schon in einer Vielzahl an Arbeiten ausgenutzt:

- Brook [Buck et al., 2004] abstrahiert GPU's als Stream Prozessoren, wobei zur Ausführung die programmierbare Grafikpipeline der GPU verwendet wird.
- CUDA [NVIDIA Corporation, 2010], ebenfalls auf dem Stream-Modell aufbauend, erlaubt die unmittelbare Programmierung von Kernels und deren Kommunikation. Die verwendete Sprache ist ähnlich zu C, erweitert die Sprache aber um Synchronisierungsmechanismen und Vektordatentypen.
- BSGP [Hou et al., 2008], basierend auf dem *Bulk Synchronous Parallel Programming* Modell abstrahiert Datenabhängigkeit, indem Kernel automatisch generiert werden und deren Datenfluss global optimiert wird.
- Auf Library-Ebene bieten High-Level Abstraktionen wie Accelerator [Tarditi et al., 2006] eingeschränkte Funktionalität, da sich die API nur auf Standard-Array-Operationen beschränkt und Kernels für bestimmte Algorithmen zum Teil nicht automatisch in optimaler Weise erzeugt werden können Hou et al. [2008].
- Basierend auf Array Operationen wurden Domain Specific Languages entwickelt, um damit die Programmierung einfacher zu gestalten und besser in bestehenden Code integrieren zu können [Chakravarty et al., 2011; Chafi et al., 2010]. In [Peercy et al., 2006] wurde für Array Operationen ein Befehlssatz entwickelt, welcher mittels Virtual-Machine implementiert wurde und somit über Array-Operationen abstrahiert.
- In Brahma [Ananth, 2010] werden Array Operationen über LINQ Expression Trees abstrahiert und Code für verschiedene GPU Sprachen erzeugt. In [Don, 2006] wurden als Fallbeispiel für F# Metaprogramming aus simplem F# Code die entsprechenden Accelerator API Calls erzeugt.
- In [Leung et al., 2009] wurde der Java JIT Compiler erweitert, um je nach Metrik Code auf der GPU auszuführen. Dieser Ansatz bringt den Aufwand mit sich, aus Java Bytecode Programmlogik zu rekonstruieren, um dann über die Parallelisierung zu entscheiden.

2.2 Implementierung funktionaler Sprachen

2.2.1 Ausführungsstrategien

Grundsätzlich unterscheidet man zwischen folgenden Ausführungsmodellen:

1. **Interpretation:** Das Programm im Lambda Kalkül, in Form des abstrakten Syntaxbaumes wird hier direkt für die Interpretation verwendet. Hierbei gibt es wiederum verschiedene Ansätze. Verschiedene Herangehensweisen werden in [Augustsson, 2000] skizziert.

2. **Als Abstrakte Maschine:** Das Programm wird auf maschinenunabhängigen Bytecode übersetzt und dieser daraufhin interpretiert oder auch per Just-In-Time-Compilation übersetzt und ausgeführt. Populäre Implementierungen sind dabei:

   - Die erste virtuelle Maschine, die eine funktionale Sprache (*Call-By-Value Lambda Calculus*) realisiert ist die SECD-Maschine [Landin, 1967].
   - Cardellis stackbasierte *Functional Abstract Machine (FAM)* Cardelli [1983].

3. **Übersetzung auf nativen Maschinencode:** Hierbei wird meistens auf eine imperative High-Level Sprache übersetzt, welche dann erst tatsächlich auf Maschinencode übersetzt wird .

Im Rahmen dieser Arbeit ist in erster Linie die Übersetzung auf imperative High-Level Sprachen von Bedeutung.

Funktionale Sprachen können aufgrund ihrer "Evaluation Strategy" grob in zwei grundlegende Gruppen unterteilt werden.

Strict Evaluation: Vor der Ausführung von Funktionen werden *alle* Argumente evaluiert. Die Reihenfolge der Auswertung der Argumente ist ein weiteres Unterscheidungsmerkmal. Als wichtiger Vertreter ist *Standard-ML* zu nennen.

Non-strict Evaluation/Lazy Evaluation: Hier werden Funktionen ausgeführt, bevor deren Argumente evaluiert worden sind. *Call-by-Name* evaluiert Argumente, sobald diese benötigt werden. *Call-by-Need* agiert wie *Call-by-Name*, wertet allerdings mehrmals verwendete Argumente nicht mehrmals aus.

Die essentielle Design-Entscheidung der "Evaluation Strategy" beeinflusst in weiterer Folge auch den Aufbau des Compilers bzw. dessen Zwischensprachen. Die Implementierung von Funktionalen Sprachen mit Lazy Auswertungsstrategie werden in Peyton Jones [1987] behandelt. Da sich F# aber "Strict Evaluation" Semantik bedient, wird auf Non-strict Evaluation hier nicht näher eingegangen.

2.2.2 Zwischenrepräsentationen

Die Wahl und Architektur der Zwischensprache spielt bei Übersetzern im allgemeinen eine Entscheidende Rolle. Bei *strict-evaluated* funktionalen Sprachen ist die Linearisierung der *Computations* besonders relevant (und die Fähigkeit der Representation dies auszudrücken). In Steele [1978] wurde *Continuation Passing Style (CPS)* erstmals als Zwischenrepräsentation verwendet. Appel [1992] bindet alle nicht trivialen Werte auf Variablen, was den Kontrollfluss explizit macht. Eine weitere häufig verwendete Zwischenrepräsenatation ist die *A-normal form (ANF)* Flanagan et al. [1993], welche sehr verwandt zu CPS ist, allerdings keine volle CPS ist und auch nicht alle Vorteile von CPS hat. Beide erwähnten Repräsentationen sind stark verwandt zu *Static Single Assignment*, wie in Kelsey [1995] gezeigt.
Der Hauptvorteil voller CPS gegenüber *A-normal form* ist die Abgeschlossenheit gegenüber Compiler-Transformationen. Beispielsweise ist CPS gegenüber $\beta - reduction\ (inlining)$ abgeschlossen (d.h. das Resultat ist wieder in CPS). $let\ x = (\lambda y.let\ z = a\ b\ in\ c)\ d\ in\ e$ wird nach naiver $beta - reduction$ zu $let\ x = (let\ z = a\ b\ in\ c)\ in\ e$, was nicht mehr in ANF ist.
Monadic Languages sind eine weitere Form der Repräsentation, die beispielsweise in MLJ Benton et al. [1998] verwendet wurde. In Kennedy [2007] werden Vorteile bezüglich Optimierung und Effizienz von *CPS* gegenüber *ANF* und *Monadic Languages* argumentiert woraufhin MLJ auch auf CPS umgestellt wurde.

2.2.3 Defunctionalization

Die Zielsprache GLSL unterstützt keine verschachtelten Funktionsdefinitionen, keine Funktionspointer aber auch keine *labels* bzw. *gotos*. Übliche Implementierungstechniken sind daher nicht direkt anwendbar.

- Etliche Implementierungen übersetzen direkt auf ASM wie etwa Appel and Macqueen [1987], Tarditi et al. [1995] generieren nativen Maschinencode oder auch wie Benton et al. [1998] auf JVM. Dabei werden erwähnte Konstrukte verwendet, die wir in unseren Zielsprachen nicht zur Verfügung haben.

- Tolmach and Oliva [1998] übersetzen auf C, aber auch ADA. Der darin vorgestellte Closure-Mechanismus kommt ohne Funktionspointer und unsichere Casts aus. Tarditi et al. [1990] übersetzen ebenfalls auf portables C99 und verwenden den aus Steele [1978] bekannten Apply-Mechanismus.

Für diese Arbeit sind also haupsächlich Techniken relevant, welche auf *Defunctionalization* basieren. Dabei werden Closures als Datentypen dargestellt und zur Ausführung von einer *apply*-Funktion gebrauch gemacht. Das folgende Beispiel skizziert[1] das Konzept.

[1] Das Beispiel weicht von den tatsächlichen Implementierungen ab und sollte nur das Konzept darstellen

```
let multFunc : int -> int -> int   = fun x y -> x * y
let mult2    : int -> int          = multFunc 2
let mult3    : (int -> int) -> int = fun f -> f 3
let mult2x3 = mult3 mult2


type Clos = Clos    of int
          | Clos2 of int * int

let apply c = match c with
               | Clos(x)    -> failwith "not enough args"
               | Clos2(x,y) -> multFunc x y

let mult2' : Clos = Clos 2
let mult3' : Clos -> int = fun (Clos x) -> apply (Clos2 (x,3))
let mult2x3' = mult3' mult2'
```

mult3 verwendet hier *mult2* als *Higher-Order*-Argument. In der *defunktionalisierten* Version *mult3'* wird lediglich ein Wert vom Typ *Clos* übergeben. Dabei wird deutlich, dass Closures als Datentypen dargestellt werden und bei der Ausführung ein Dispatch auf die tatsächliche Funktion stattzufinden hat. Erstaunlicherweise ist dies auch für "dynamische erzeugte Closures" möglich. Dazu sei auf Danvy and Nielsen [2001] verwiesen.

2.2.4 Lambda-Lifting

Um *Defunctionalization* durchführen zu können, dürfen keine freien Variablen in den Ausdrücken vorkommen (oder man zieht die freien Variablen ebenfalls in die *Closure-Datenstruktur*). *Lambda-Lifting* Johnsson [1985] erweitert Funktionen um zusätzliche Argumente, welche die freien Variablen darstellen und somit nicht mehr *frei* auftreten. Nach *Defunctionalization* und *Lambda-Lifting* erhält man ein First-Order-Programm bestehend aus High-Level-Definitionen.

2.2.5 source-to-source Übersetzung

In Tolmach and Oliva [1998] wird eine Source-to-Source Übersetzung von ML nach Ada behandelt. Da sich F# in die ML-Famlie einordnen lässt und die Zielsprache, Ada hinsichtlich dem Typsystem einfach und sauber ist (frei von Casts) hat diese Publikation besondere Relevanz. Ada hat zudem noch ähnliche Limitierungen, wie sie in Shadersprachen vorhanden sind. Hier sei hauptsächlich auf Memory-Management und Pointer Arithmetik hingewiesen.

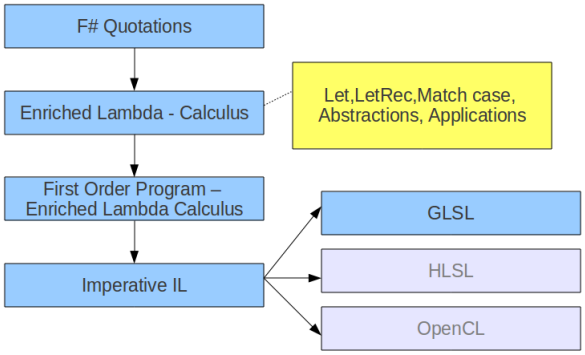


Abbildung 2: Schematischer Ablauf der Übersetzung.

3 Implementierung des Compilers

3.1 Architektur des Compilers

In 1 werden die Phasen der Übersetzung schematisch dargestellt. Im folgenden werden die einzelnen Phasen und die verwendeten Zwischensprachen beschrieben.

Optimierungen erstrecken sich über alle Phasen und werden im jeweiligen Übersetzungsschritt beschrieben.

3.1.1 F#-Code

Mittels F#-Quotations, ist es möglich, beliebigen[1] (statisch korrekten) F#-Code zur Laufzeit in den Abstrakten Syntax-Baum zu konvertieren. Die Spezifikation der Sprache ist unter Syme [2010] zu finden.

Beispiel:

```
let bsp = <@ (fun x -> x + 1) @> ;;

val bsp : Quotations.Expr<(int -> int)> =
  Lambda (x,
          Call (None, Int32 op_Addition[Int32,Int32,Int32](Int32, Int32),
                [x, Value (1)]))
```

Dies würde folgenden Syntax-Baum entsprechen:

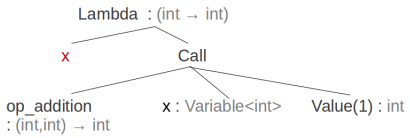


Abbildung 3: SyntaxTree eines einfachen Ausdrucks

[1] Äußere Ausdrücke dürfen nicht generisch sein, dazu müssen gegebenenfalls Typanotationen hinzugefügt werden Microsoft [2010]

Wie in der Skizze angedeutet, wird auch die Semantische Analyse vom F#-Compiler übernommen. Das heißt, auch die für F# nicht triviale Typinferenz wird von Quotations erledigt.
Um nicht jede Funktion, welche im Eingangsprogramm verwendet werden soll als Quotation zu definieren ist es auch möglich, mittels Reflection den Syntax-Baum von Funktionen zu erhalten. Dies geschieht über ein Attribut, welches über die Funktion gesetzt wird - dieses Attribut aktiviert die Möglichkeit die Funktion in Form des Syntaxbaumes zur Laufzeit zu erhalten.

Erneute Übersetzung Anzumerken ist auch, dass Werte vom Typ Expression (also Syntaxbäume, welche durch Reflection oder Quotations entstanden sind) auch wieder in ausführbaren Code übersetzt werden können. Dies ist vor allem für partielle Ausführung und diverse Optimierungen sinnvoll.

Eine detaillierte Übersicht über Implementierung,Theorie und Beispiele von Quotations wird in [Don, 2006] gegeben.

3.1.2 Funktionale Zwischensprache - TypedCore

Die Quellsprache, das Resultat der vorigen Phase ist durch folgende abstrakte Syntax in EBNF gegeben:

⟨*CoreExpr*⟩ ::= 'λ' ⟨*Identifier*⟩ '$\rightarrow$' ⟨*CoreExpr*⟩
| 'CApp' ⟨*CoreExpr*⟩ ⟨*CoreExpr*⟩
| 'CApps' ⟨*CoreExpr*⟩ ⟨*CoreExpr*⟩+
| 'CLet' ⟨*Identifier*⟩ '=' ⟨*CoreExpr*⟩ 'in' ⟨*CoreExpr*⟩
| 'CLetRec' ⟨*Identifier*⟩ '=' ⟨*CoreExpr*⟩ 'in' ⟨*CoreExpr*⟩
| 'CIf' ⟨*CoreExpr*⟩ 'then' ⟨*CoreExpr*⟩ 'else' ⟨*CoreExpr*⟩
| 'CIntrensicCall' ⟨*Identifier*⟩
| ⟨*Value*⟩

⟨*Value*⟩ ::= 'CConstant' ⟨*Literal*⟩
| 'CTup' ⟨*CoreExpr*⟩*
| 'CVariable' ⟨*Identifier*⟩
| 'CProjection' ⟨*CoreExpr*⟩ ⟨*Number*⟩

Die Sprache erweitert den untypisierten Lambda-Kalkül um folgende syntaktische Elemente, deren Semantik informell definiert wird:

CApps Funktionen mit mehreren Argumenten der Form $\lambda x_0 \rightarrow \lambda x_1 \rightarrow x_n \rightarrow e$ werden (bei voller Reduktion) anstatt Hintereinanderausführung von Applications $((..(ex_0)x_1)..x_n))$ als $CApps(e, [x_0..x_n])$ formuliert. Diese Vereinfachung ist reine Optimierung, um volle Reduktionen kompakter darzustellen.

CLet Ein Ausdruck der Form *let var = expr in body* bindet das Binding (var,expr) im lokalen lexikalischen Scope von body und ist semantisch äquivalent zu $CApp((\lambda var \rightarrow body), expr)$

CLetRec Die gebundene Variable ist sowohl in body, als auch in expr selbst verfügbar.

CIf *CIf expr then expr1 else expr1* wertet *expr* vom Type Bool aus. Evaluiert expr zu True wird expr1 evaluiert, anderenfalls expr1. Expr1 und Expr2 sind vom selben Typ.

CIntrensicCall Bildet Ausdrücke ab, die von der Zielsprache direkt unterstützt werden. Beispielsweise wird numerische (oder vektorielle) Multiplikation als *CApps(CIntrensicCall(mult),x,y)* dargestellt.

Implementiert wurde die gegebene Sprache als Modellierung durch Discriminated Unions. Optisch wird der ML-Typ sehr ähnlich der angeführten Grammatik. Aus praktischen Gründen werden Ausdrücke immer als Tupel aus Ausdruck und dessen Typ dargestellt. Daraus ergibt sich eine leicht andere Darstellung.

```
                                                                                    1
type CoreExpr = CLambda       of Identifier * TypedCore                             2
              | CApp          of TypedCore  * TypedCore                             3
              | CApps         of TypedCore list                                     4
              | CIf           of TypedCore * TypedCore * TypedCore                  5
              | CLet          of Identifier * TypedCore * TypedCore                 6
              | CProjection of TypedCore * int                                      7
              | CIntrensicCall of string                                            8
              | CValue                                                              9
                                                                                    10
and type CValue = | Constant    of obj                                              11
                  | CTup         of Tuple                                          12
                  | CVariable    of Identifier                                     13
                                                                                    14
and type TypedCore = CoreExpr * CType                                               15
                                                                                    16
type PrimitiveType = Float | Int |...                                               17
                                                                                    18
type CType = Fun     of CType * CType                                               19
           | FunT    of CType list * CType                                          20
           | PrimitiveType of PrimitiveType                                         21
           | GenericType   of CType                                                 22
           | HGenericType  of CType list                                            23
           | CustomType    of string                                                24
           | Bottom                                                                 25
```

Typen sind auch mittels Discriminated Union implementiert. Dabei werden Funktionstypen, wie auch generische Typen abgebildet.

Funktions Typen: Funktionstypen liegen *curried* vor. Der Ausdruck *let foo x y = x * y* hat beispielsweise den wohlgeformten Typ: *Fun (PrimitiveType Float,Fun (PrimitiveType Float,PrimitiveType Float)))*. Ähnlich zu *CApps* können Funktionstypen mit mehreren Argumenten mittels FunT dargestellt werden. Für das Beispiel hieße das: *FunT([PrimitiveType Float;PrimitiveType Float],PrimitiveType Float)*

Abbildung 4: Schematischer Ablauf der Übersetzung.

Generische Typen: GenericType und HGenericType bilden Typen ab, welche mittels weitere Typen definiert sind. Beispielsweise bildet der Typ *HGenericType(PrimitiveType Float,PrimitiveType Float, PrimitiveType Float)* einen 3-dimensionalen Vektor ab.

Benutzerdefinierte-Typen: Mittels CustomType können Strukturen abgebildet werden.

3.1.3 Imperative Zwischensrpache - IIL

Um den Compiler möglichst allgemein zu gestalten und auch andere Backends als GLSL wie etwa HLSL, CUDA oder OpenCL zu unterstützen, wurde aus den "First-Order HighLevel Funktionsdefinitionen" nicht direkt GLSL erzeugt, sondern in eine Imperative Zwischensprache übersetzt. Diese Sprache abstrahiert über die genannten anderen Zielsprachen und stellt daher eine simple, imperative und auf Statements basierende Sprache dar.

```
                                                                                    1
type IExpr = IConstant of obj * CType                                               2
           | ILhsVariable of IVariable                                              3
           | IBottom                                                                4
           | IUnionType of string * (IExpr list)  * CType                           5
           | IIntrensicCall of string * (IExpr list) * CType                        6
           | ICall of string * (IExpr list) * CType                                 7
                                                                                    8
and IVariable = IVariable of string * CType                                         9
                                                                                    10
and IStatement = IAssignment of IVariable * IExpr                                   11
               | IReturn of IExpr                                                   12
               | INop                                                               13
               | IFunc   of string * (IExpr list) * IStatementList                  14
and IStatementList = IStatement list                                                15
```

Ein Programm besteht aus einer Liste an Statements, die Funktionsdefinitionen darstellen. Diese Funktionsdefinitionen beinhalten wiederum eine Liste an Statements, welche mit einem IReturn abschließen.

3.1.4 Zielsprache GLSL

Die volle Spezifikation der Zielsprache ist in John Kessenich [2011] zu finden.

3.2 Übersetzung in die Imperative Zwischensprache

Die Übersetzung wird mit Hilfe eines Beispiels demonstriert. Dabei wird folgendes F# Programm[2] verwendet:

```
<@ let pi = 3.14                                                                    1
   let geometry (u:float)  =                                                        2
       let u = cos (u * pi)                                                         3
       (u,0.0,0.0)                                                                  4
   in geometry @>                                                                   5
```

3.2.1 Transformationen auf Quotations.Expr

Die Transformation von F#-Code zur *Quotations.Expr*-Instanz (*Expr* ist die Klasse zur Beschreibung von Syntaxbäumen und wird im Namensraum *Quotations* definiert) wird von Core F# Libaries durchgeführt. Ergebnis davon ist ein typisierter AST:

```
Let (pi, Value (3.14),
     Let (geometry,
          Lambda (u,
                  Let (u,
                       Call (None, Double Cos[Double](Double),
                             [Call (None,
                                    Double op_Multiply[Double,Double,Double](Double, Double),
                                    [u, pi])]),
                       NewTuple (u, Value (0.0), Value (0.0)))), geometry))
```

Programme, welche als Quotations.Expr Instanz vorliegen eignen sich für Optimierungen, da mittels "Compile" Teile des Programms im Kontext der VM ausgeführt werden können.

3.2.2 Transformationen auf TypedCore IL

Die Ergebnisse der Zwischenschritte werden aus Übersichtlichkeitsgründen als F# - Code angezeigt. Die Code-Listings entsprechen also nicht direkt der Datenstruktur, auf die die Transformationen ausgeführt werden, sondern einer für Menschen einfach lesbaren Form. Weiters werden in jeder Transformation, in welcher Namen erzeugt oder modifiziert werden, spezielle Präfixes vor die Namen gesetzt, um die Schritte besser nachvollziehen zu können.

Vergabe eindeutiger Identifier

Dadurch werden Namenskollisionen von vornherein ausgeschlossen. Weiters hat sich herausgestellt, dass eindeutige Namen in den nachfolgenden Phasen Vorteile bezüglich Verständlichkeit und Komplexität der Algorithmen bringen. Bezeichner werden mit Präfix versehen, um im Zwischencode zu verdeutlichen, dass es sich hier um modifzierte Namen handelt. Als Suffix wird der Verschachtelungsgrad der verdeckten Variable verwendet. $let\ x = let\ x = e$ wird zu $let\ x_0 = letx\_1 = e$.
Dabei wird der Graph durchlaufen und eine Tabelle mit vergebenen Namen mitgeführt. Kommt es zu einem Namenskonflikt, wird für die jeweilige Variable ein frischer Bezeichner gewählt und deren Vorkommnisse entsprechend umbenannt.

```
let $pi_0 = 3.14
let $geometry_0 = fun $u_0 ->
        let $u_1 = cos ($u_0 * $pi_0)
        ($u_1,0.0,0.0)
in geometry
$
```

[2] Es wurde absichtlich (mitunter aus Platzgründen) ein triviales Beispiel gewählt, um diverse Transformationen besser visualisieren zu können.

Transformation zur ANF In 2.2.2 wurde CPS und ANF behandelt. Verwendung von CPS als Zwischenrepräsentation hat aufgrund der Abgechlossenheit gegenber $\beta - reduction$ Vorteile für Transformationen in späteren Compiler-Phasen.
Allerdings birgt CPS einige Nachteile: CPS Terme erscheinen komplexer und wesentlich weniger kompakt. Weiters bringt CPS Kosten für die Allokation von Closures mit sich. Um diese Nachteile loszuwerden wurden verschiedenste Algorithmen vorgestellt. In Kennedy [2007] wird argumentiert, dass diese Nachteile keine tatsächlichen Nachteile sind - tatsächlich wird sogar präsentiert, wie der Zwischencode dadurch lesbarer und durch Trennung administrativer und ursprüngliche Lambdas zu klareren Code führen kann.
Aus Sicht des Autors ist eine CPS-Transformation im Kontext des GPU-Compilers allerdings wenig zielführend, da sich der erhebliche Mehraufwand zur Reduktion administrativer Closures im Vergleich zur ANF-Repräsentation (und dessen Renormalisierung nach Transformationen) nicht rentiert, wenn man bedenkt, dass Closures auf der Zielhardware nur verhältnismäßig ineffizient dargestellt werden können und jeder unnötige administrativer Redex die Effizienz entschieden verringern kann.

```
let $pi_0 = 3.14                                                    1
let $geometry_0 = fun $u_0 ->                                       2
    let $anf0 = $u_0 * $pi_0                                        3
        let $u_1 = cos $anf0                                        4
        ($u_1,0.0,0.0)                                              5
in geometry                                                         6
$                                                                   7
```

Abbildung 5: ANF-Transformation, angewendet auf das laufende Beispiel

Defunctionalization

Die Lambda-Audrücke können in dieser Phase in beliebiger Form vorkommen. Partiell ausgewertete Funktionen können als Argumente auftauchen. Hierbei spricht man von *Closures as first class citizens*. Beispielsweise könnte für die Definition *let mult a b = a * b in let mult2 = mult 2 in mult2 2* nicht direkt Code erzeugt werden. In der ersten Version des Übersetzers fand die explizite Darstellung der Closures erst nach *Lambda-Lifting* und *Exktraktion von Top-Level Definitionen* statt . Es hat sich jedoch gezeigt, dass die Übersetzung sauberer und effizienter erfolgen kann, wenn Closures welche als Argumente vorkommen schon in den Lambda-Ausdrücken in eine explizite Darstellung konvertiert werden. Der verwendete Algorithmus arbeitet konzeptuell wie in 2.2.3 beschrieben. Die Implementierung gestaltet sich als besonders aufwändig, da die notwendigen Transformationen die Typen einzelner Ausdrücke verändern und somit eine Kette an weiteren Transformationen auslösen (um die Typkonsistenz wieder herzustellen). Der eingesetzte Algorithmus verfährt iterativ, indem *Let-Audsdrücke*, dessen gebundener Wert als Higher-Order-Argument vorkommt als *Record*-Typ dargestellt wird und die Verwendungen der Variable entsprechend angepasst werden.

Lambda-Lifting

Lambda-Lifting wurde wie in Johnsson [1985] *Section 3* implementiert und hat kubische Laufzeit im Bezug auf die Anzahl der zu behandelnden *Let/LetRecs*. Es sei angemerkt, dass es effizientere Algorithmen gibt, wie etwa in Danvy and Schultz [2002] gezeigt wurde. Da die Eingabeprogramme relativ klein sind, wurde eine derartige effizientere Lösung nicht implementiert.

```
let $pi_0 = 3.14
let $geometry_0 =
  fun $$pi_0 ->
    fun $u_0 ->
      let $anf0 = $u_0 * $$pi_0
          let $u_1 = cos $anf0
          ($u_1,0.0,0.0)
in (geometry $pi_0)
$
```

Abbildung 6: Lambda-Lifting, angewendet auf das laufende Beispiel

Exktraktion von Top-Level Definitionen

Nach der vorigen Phase liegt das Programm grundsätzlich schon als Ausdruck bestehend aus High-Level Funktionen vor. Aus Implementierungssicht wurde hier jede Funktion extrahiert und in linearer Form abgelegt (d.h. als Liste von Funktionen).

```
let $pi_0 = 3.14
let $anf0 $u_0 $$pi_0 = $u_0 * $$pi_0
let $u_1 $anf0 = cos $anf0
let $geometry_0 $$pi_0 $u_0 =
    let $anf1 = ($anf_0 $u_0 $$pi_0)
    ($u_1 $anf1,0.0,0.0)
let $main = geometry $pi_0
```

Abbildung 7: Extraktion von Top-Level Definitionen angewendet auf das laufende Beispiel

Uncurrying - Volle Reduktionen

Kommen im Programm nur mehr volle Funktionsauswertungen vor, wird eine Optimierung vorgenommen. *let mult a b = a * b in ((mult 2) 2)* kann mittels zweistelliger Funktion effizienter dargestellt werden *let mult a b = a * b in mult(a,b)*. Der verwendete Algorithmus ist als Globale-Analyse zu sehen - Alle Funktionen und deren Argumente sind bekannt[1]. Konkret Bedeutet die Transformation die Ersetzung aller *CApp(...CApp(f,arg0)...)* durch *CApps(arg0,..,argN)*.

3.2.3 Transformationen auf IIL

Die Transformation in die IIL-Zwischensprache gestaltet sich als sehr einfach, da die vorigen Phasen bereits durch ANF linearisiert und auch in SSA vorliegen (ausgehend von 3.2.2). Weiters liegen bereits High-Level-Definitionen vor, dessen Argumente und deren Typen auch bekannt sind. Definitionen dürfen außerdem rekursiv sein, was *Tail-Call-Optimization* bzw. die Transformation rekursiver Calls in

[1] Anonyme Lambdas wurden mit Namen versehen und sind Top-Level-Definitionen

```
map                                                                                                                                                    1
    [("$$main",                                                                                                                                        2
      ([],                                                                                                                                             3
       (CApps                                                                                                                                          4
          [(CVariable "$geometry_0",                                                                                                                   5
            ...Types ommited...                                                                                                                        6
            (CVariable "$pi_0", PrimitiveType Float)],                                                                                                 7
         Fun (PrimitiveType Float,                                                                                                                     8
           HGenericType [PrimitiveType Float; PrimitiveType Float; PrimitiveType Float]))));                                                           9
     ("$geometry_0",                                                                                                                                   10
      ([("$pi_0", PrimitiveType Float); ("$u_0", PrimitiveType Float)],                                                                                11
       (CTup (Tuple                                                                                                                                    12
              [(CApps                                                                                                                                  13
                  [(CVariable "$u_1", PrimitiveType Float);                                                                                            14
                   (CVariable "$pi_0", PrimitiveType Float);                                                                                           15
                   (CVariable "$u_0", PrimitiveType Float)],                                                                                           16
                PrimitiveType Float); (Constant 0.0, PrimitiveType Float);                                                                             17
               (Constant 0.0, PrimitiveType Float)]),                                                                                                  18
         ... Types ommited...                                                                                                                          19
     ("$pi_0", ([], (Constant 3.14, PrimitiveType Float)));                                                                                            20
     ("$u_1",                                                                                                                                          21
      ([("$pi_0", PrimitiveType Float); ("$u_0", PrimitiveType Float)],                                                                                22
       (CLet                                                                                                                                           23
          ("$anf1",                                                                                                                                    24
           (CApps                                                                                                                                      25
              [(CIntrensicCall "*", ...Types ommited...                                                                                                26
               (CVariable "$u_0", PrimitiveType Float);                                                                                                27
               (CVariable "$pi_0", PrimitiveType Float)], PrimitiveType Float),                                                                        28
           (CApps                                                                                                                                      29
              [(CIntrensicCall "cos",                                                                                                                  30
                Fun (PrimitiveType Float,PrimitiveType Float));                                                                                        31
               (CVariable "$anf1", PrimitiveType Float)], PrimitiveType Float)),                                                                       32
        PrimitiveType Float)))]                                                                                                                        33
$                                                                                                                                                      34
```

Abbildung 8: Uncurrying, angewendet auf das laufende Beispiel

Kontrollstrukturen wie Schleifen bislang nicht notwendig macht. Die Transformation ist mittels einer einfachen Traversal implementiert.

3.3 Code Generierung - GLSL

Der gesamte Übersetzungsprozess wird mittels eines nicht-trivialen Beispiels demonstriert. Das Ausgangsprogramm modelliert einen VertexShader, welcher eine Kugel modelliert. Dazu werden Kugelkoordinaten aus einem ([0,1],[0,1]) Netz erzeugt (UV-Parametrisierung).

```
let geometry =                                                  1
    <@ let pi = 3.14                                            2
       let geometry u v =                                       3
                          let theta =  u * pi                   4
                          let phi = 2.0 * pi * v                5
                                                                6
                          let sinCos (u:float) =                7
                                (sin theta) * (cos phi)         8
                                                                9
                          let x = sinCos 0.0                    10
                          let y = (sin theta) * (sin phi)       11
                          let z = cos theta                     12
                          (x,y,z)                               13
        in geometry @>                                          14
```

9 zeigt das erzeugte Programm, lauffähig in GLSL:

3.4 Limitierungen

Die derzeitige Implementierung des Übersetzers ist zum Teil noch sehr limitiert. Einige Anregungen für Zukünftige Arbeit sind:

```
float _z_0(float _u_0);
float _y_0(float _u_0, float _v_0);
float _x_0(float _u_0, float _v_0);
float _theta_0(float _u_0);
float _sinCos_0(float _u_0, float _v_0, float _u_1);
const float _pi_0 = 3.140000;
float _phi_0(float _v_0);
vec3 _geometry_0(float _u_0, float _v_0);
//--------------------

//----Functions------
float _z_0(float _u_0)
{
        float _anf1 = _theta_0(_u_0);
        return cos(_anf1);
}
float _y_0(float _u_0, float _v_0)
{
        float _anf2 = sin(_phi_0(_v_0));
        float _anf1 = sin(_theta_0(_u_0));
        return _anf1 * _anf2;
}
float _x_0(float _u_0, float _v_0)
{
        return _sinCos_0(_u_0, _v_0, 0.000000);
}
float _theta_0(float _u_0)
{
        return _u_0 * _pi_0;
}
float _sinCos_0(float _u_0, float _v_0, float _u_1)
{
        float _anf2 = cos(_phi_0(_v_0));
        float _anf1 = sin(_theta_0(_u_0));
        return _anf1 * _anf2;
}
float _phi_0(float _v_0)
{
        float _anf1 = 2.000000 * _pi_0;
        return _anf1 * _v_0;
}
vec3 _geometry_0(float _u_0, float _v_0)
{
        return vec3(_x_0(_u_0, _v_0), _y_0(_u_0, _v_0), _z_0(_u_0));
}
//--------------------
void main()
{
    vec2 uv = gl_Vertex.xy;
    vec4 p = vec4(_geometry_0(uv.x,uv.y),1.0);
        gl_Position = gl_ModelViewProjectionMatrix * p;
    gl_FrontColor  = vec4(1.0,0.0,0.0,1.0);
}
```

Abbildung 9: Endresulat der Übersetzung

Rekursion ist in GLSL nicht erlaubt - primitive Rekursion könnte mittels effizienter Stack-Implementierung in Schleifen umgewandelt werden. Tail-Call-Optimization, essentiell für Continuation-Passing-Style Programmierung wäre eine notwendige Erweiterung.

CUDA/OpenCL Rekursion und Closures könnten in diesen Zielsprachen effizienter formuliert werden, da Funktionspointer und Rekursion erlaubt ist.

Modules Derzeit findet der ganze Übersetzungsprozess auf dem gesamten Programm statt bzw. muss bei jeder Änderung das ganze Programm übersetzt werden. F#-Module werden nicht unterstützt.

Typ-System Derzeit ist es nicht möglich Typ-Definitionen zu übersetzen. Das Typsystem der Zielsprache ist derzeit nicht erweiterbar (obwohl die Zwischensprache *TypedCore* sehr flexible Typen anbietet).

Statische-Optimierung Statische Argumente für Programme könnten dazu verwendet werden, Teile der Programme zur Compilezeit auszuführen. Die F#-Infrastruktur könnte wie in 3.1.1 dazu verwendet werden.

Effiziente-Übersetzung Für diverse Transformationen (z.B.: *Lambda-Lifting*) existieren effizientere Algorithmen.

Attribute-Grammars Einsatz eines Attribute-Grammar Systems könnte die Code-Complexität erheblich senken und weitere Optimierungen und Transformationen erleichtern.

4 Resultate

4.1 Functional Graphics - ShaderIDE

4.1.1 Motivation

In Realtime-Applikationen werden Shader-Programme zunehmend wichtiger, getrieben durch flexible API's die es ermöglichen große Teile der Rendering-Pipeline frei zu programmieren. GLSL John Kessenich [2011], Cg Corporation [2011] und HLSL Microsoft [2011] werden in der Regel dazu verwendet[2]. Aus sprachtheoretischer Sicht unterscheiden sich die Sprachen nur subtil, sind aber nicht zueinander kompatibel. Zusätzlich verändern sich die Sprachen schnell - d.h. Code muss wiederholt portiert werden. Weiters fehlen wichtige Abstraktionsmittel wie OOP[1]. Shaderprogramme implementiert in F# bieten einige Vorteile:

- Hardware-Unabhängigkeit
- Hohes Abstraktionsniveau: Funktionalität wie OOP Interfaces bzw. Higher-Order-Functions
- Hohe Code-Wiederverwendbarkeit: F# bzw. ML Codebasis kann weiterverwendet werden, Potentielle Sprachänderungen erfordern keine Modifikation der Shader-Programme.
- Zur Entwicklung können die gewohnen IDEs verwendet werden und der Code auch über übliche Debugging Tools optimiert/debugged.

Die entwickelte Applikation dient allein dazu, schnell generierten Code inspizieren zu können und Analysesn durchzuführen (siehe 4.1.3).

4.1.2 Beschreibung der Applikation

Die Applikation besteht aus einer Echtzeitvorschau, einem Eingabefeld für Shaderprogramme (in F#) und einem Ausgabefeld für den generierten GLSL Code. Screenshots des Programmes werden in 10 gezeigt.

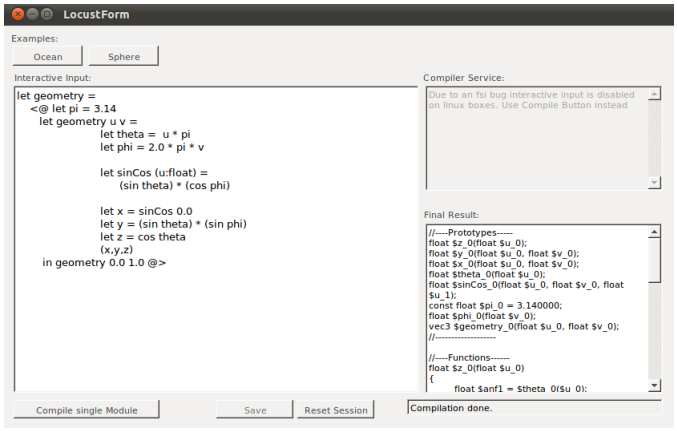
4.1.3 Performance - Analyse

Getestet wurde die Berechnung einer 3D-Surface. Dies wurde realisiert durch Sampling über ein $[0,1] \times [0,1]$ 2D-Intervall. Dabei wurden 1024^2 uniform verteilte Sampling-Punkte verwendet. Konkret wurden als Testprogramme die in 3.3 vorgestellten Programme verwendet. Pro Vertex müssen 2 * 32bit floats und 1 * 32bit integer (für indices) übertragen werden. Das ergibt eine Datenmenge von 12MB.

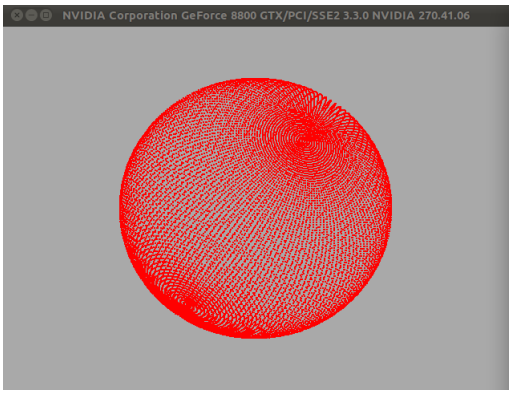
Getestet wurde mit Intel(R) *Core(TM)2 Quad CPU Q6600 @ 2.40GHz*, 4GB RAM, Geforce8800GTX 512MB, Ubuntu Natty 11.04 (Kernel: 2.6.38), Mono 2.6.7, Übersetzung der C# und F# Programme

[1] Cg und HLSL unterstützen eine limitierte Form von OOP-Interfaces

[2] Grafik-APIs unterstützen sehr wohl direkte ASM-Programmierung - diese ist allerdings nicht mehr praktikabel.



(a) Screenshot des ShaderEditors



(b) Screenshot der Echtzeitvorschau des ShaderEditors

Abbildung 10: Screenshots des ShaderEditors

wurde mit den .NET4.0 Compilern unter *Release Configuration* durchgeführt. Um möglichen Bias durch den mono-Jitter auszugleichen wurden jeweils auch Messungen unter Windows 7 mit .NET VM (3.5) durchgeführt. In den folgenden Tests werden diese Messungen nur bei signifikanter Differenz zur Mono VM angegeben.

Betrachten wir zunächst die reine Berechnungszeit (Zeit für Transformation und Sampling über das 2D-Intervall):

ShaderCode	Handoptimiert	Aus F# übersetzt
Mono	133	133
.NET	118	117

Tabelle 1: Messungen der reinen Ausführungszeit (in ms)

Überraschenderweise unterscheidet sich die Ausführungszeit zwischen dem händisch optimierten und dem komplexeren automatisch generierten Code nicht innerhalb der Messgenauigkeit. Um diese Tatsache näher zu untersuchen, wurden die Shader mittels AMD's GPU-ShaderAnalyzer übersetzt und analysiert.

ShaderCode	Handoptimiert	Aus F# übersetzt
RadeonHD 6970 ASM	13(2)	13(2)
RadeonHD 2400 ASM	11(11)	11(11)

Tabelle 2: Analyse des generierten GPU Assemblies (ALU-Instructions/Cycles)

Die GLSL Compiler/Optimizer optimieren offenbar so aggressiv, dass der anfangs komplexere Code hinsichtlich Performance keine Auswirkungen mehr hat. Um die Aussage nicht nur für AMD GLSL

Implementierung[3] treffen zu können, wurden die Programme auf HLSL übersetzt und die Analysen auch dort durchgeführt.

ShaderCode	Handoptimiert	Aus F# übersetzt
HLSL ASM/IL-*O0*	25	78
HLSL ASM/IL-*O1*	25	25
HLSL ASM/IL-*O3*	25	25

Tabelle 3: Analyse des generierten HLSL Assembly-Codes (Instructions)

Hier ist zu erkennen, dass schon die niedrigste Optimierungsstufe, alle Nachteile des automatisch erzeugten Codes weg-optimiert. Aus Sicht der reinen Ausführungszeit überzeugt der F# GPU Compiler in den durchgeführten Tests. Nun untersuchen wir die Performance der nicht trivialen Vorbereitung - schließlich müssen u.a. die Daten in den Grafikspeicher geladen werden.

Als erstes analysieren wir die Übersetzungszeit von GLSL in tatsächlich auf der GPU ausführbaren Code. Diese Übersetzung erfolgt direkt im Treiber und ist daher stark Hardware-abhängig im Bezug auf Übersetzungszeit.

ShaderCode	Handoptimiert	Aus F# übersetzt
mono	11	13
.net	4	7

Tabelle 4: Übersetzungszeit von GLSL in ausführbaren GPU - Code (in ms)

Wie erwartet verbringt der Treiber mehr Zeit bei der Übersetzung in nativen GPU Code. Für DirectX ist dies kein Problem, da diese Übersetzung nicht zur Laufzeit ausgeführt werden muss - in OpenGL kann diese Übersetzungszeit für große Shader ein Problem werden. Für realistische Programme sollte diese Limitierung kein Problem werden, wenn man sich vor Augen führt, dass in unserem Test-Setup ca. 70 relativ komplexe Shader übersetzt werden.

Als nächtes analysieren wir die GPU-Uploads, d.h. die Zeit die notwendig ist, um die Daten für die GPU vorzubereiten und in den schnellen GPU Speicher zu laden.

GPU-Upload	ms
mono	96
.net	563

Tabelle 5: Vorbereitung der Daten + GPU-Upload

Bisher konnten wir keine Erklärung für die längere Preprocessing Zeit unter .NET finden. Mit diesen Zahlen lässt sich zeigen, ab wann sich der Upload auf die GPU rechnet.

[3]Da die Tests auf NVIDIA Hardware ausgeführt wurden, ist nicht auszuschließen, dass der analysierte Code nicht mit dem ausgeführten Code übereinstimmt, die Ergebnisse der Laufzeitanalyse decken sich allerdings auch mit den Ergebnissen des AMD GLSL Compilers.

Ausführung auf	CPU	GPU (Upload + Download)
Einmalige Ausführung	1300	296
10fache Ausführung	16500	1514

Tabelle 6: Berechnung der Oberfläche - GPU vs. CPU (Messungen in ms)

Bei 10facher Ausführung wurden die Transformationsparameter nach jeder Berechnung verändert. Die Aussagekraft dieses Tests ist kritisch zu betrachten - schließlich handelt es sich hier um ein konstruiertes Beispiel. Jedoch ist zu erkennen, dass schon bei einmaliger Ausführung für rechenintensive Operationen die Benützung der GPU vorteilhaft ist.

5 Zusammenfassung

In dieser Arbeit wurde die Implementierung einer funktionalen Sprache behandelt, mit dem Ziel den generierten Code effizient auf *Graphics Processing Units (GPUs)* ausführen zu können. Die Programmierung von GPUs birgt eine besondere Herausforderung für Software-Entwickler, da nicht die gewohnten Entwicklungshilfsmittel verwendet werden können - das vorgestellte Programm verbindet die Verwendung gewohnter Entwicklungsumgebungen mit der hohen Ausführungsgeschwindigkeit von GPUs. Es wurde gezeigt, dass der generierte Code vergleichbar effizient ausgeführt wird, wie händisch optimierter Code. Der vorgestellte Übersetzer motiviert verschiedenste Anwendungen wie etwa *General Purpose Compuation on GPUs (GPGPU)*, wie auch den Einsatz im Realtime-Rendering als Shader-Compiler.

6 Literaturverzeichnis

Balasubramania Ananth. Brahma, 2010. URL http://brahma.ananthonline.net/.

Andrew W. Appel. Compiling with continuations. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7.

Andrew W. Appel and David B. Macqueen. A Standard ML Compiler. In Functional Programming Languages and Computer Architecture, pages 301–324. Springer-Verlag, 1987.

Lennart Augustsson. Lambda-calculus cooked four ways, 2000. URL www.augustsson.net/Darcs/Lambda/top.pdf.

Nick Benton, Nick Benton, Andrew Kennedy, Andrew Kennedy, George Russell, and George Russell. Compiling Standard ML to Java Bytecodes. pages 129–140. ACM Press, 1998.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. ACM Trans. Graph., 23: 777–786, August 2004. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1015706.1015800. URL http://doi.acm.org/10.1145/1015706.1015800.

Luca Cardelli. The Functional Abstract Machine. Bell Labs Technical Journal, 1983.

Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications, OOPSLA '10, pages 835–847, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: http://doi.acm.org/10.1145/1869459.1869527. URL http://doi.acm.org/10.1145/1869459.1869527.

Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In Proceedings of the sixth workshop on Declarative aspects of multicore programming, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0486-3. doi: http://doi.acm.org/10.1145/1926354.1926358. URL http://doi.acm.org/10.1145/1926354.1926358.

Chronos Group. The OpenCL Specification, 2010. URL www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

NVIDIA Corporation. Cg Language Specification, 2011. URL http://http.developer.nvidia.com/Cg/Cg_language.html.

Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming, PPDP '01, pages 162–174, New York, NY, USA, 2001. ACM. ISBN 1-58113-388-X. doi: http://doi.acm.org/10.1145/773184.773202. URL http://doi.acm.org/10.1145/773184.773202.

Olivier Danvy and Ulrik Pagh Schultz. Lambda-lifting in quadratic time. In Proceedings of the 6th International Symposium on Functional and Logic Programming, FLOPS '02, pages 134–151, London, UK, 2002. Springer-Verlag. ISBN 3-540-44233-2. URL http://dl.acm.org/citation.cfm?id=646191.683714.

Syme Don. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In Proceedings of the 2006 workshop on ML, ML '06, pages 43–54, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9. doi: http://doi.acm.org/10.1145/1159876.1159884. URL http://doi.acm.org/10.1145/1159876.1159884.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. SIGPLAN Not., 28:237–247, June 1993. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/173262.155113. URL http://doi.acm.org/10.1145/173262.155113.

Qiming Hou, Kun Zhou, and Baining Guo. BSGP: bulk-synchronous GPU programming. ACM Trans. Graph., 27:19:1–19:12, August 2008. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1360612.1360618. URL http://doi.acm.org/10.1145/1360612.1360618.

Intel John Kessenich. The OpenGL Shading Language, 2011. URL http://www.opengl.org/registry/doc/GLSLangSpec.4.10.6.clean.pdf.

Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. pages 190–203. Springer-Verlag, 1985.

Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. SIGPLAN Not., 30:13–22, March 1995. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/202530.202532. URL http://doi.acm.org/10.1145/202530.202532.

Andrew Kennedy. Compiling with continuations, continued. SIGPLAN Not., 42:177–190, October 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1291220.1291179. URL http://doi.acm.org/10.1145/1291220.1291179.

Alan Leung, Ondřej Lhoták, and Ghulam Lashari. Automatic parallelization for graphics processing units. In Principles and Practice of Programming in Java, pages 91–100, 2009. doi: 10.1145/1596655.1596670.

Microsoft. Automatic Generalization (F#), 2010. URL http://msdn.microsoft.com/en-us/library/dd233183.aspx.

Microsoft. HLSL Reference, 2011. URL http://msdn.microsoft.com/en-us/library/bb509638%28v=VS.85%29.aspx.

NVIDIA Corporation. NVIDIA CUDA Programming Guide, 2010. URL http://tinyurl.com/6dl36l9.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krueger, Aaron Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. Computer Graphics Forum, 26(1):80–113, 2007. URL http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x.

Mark Peercy, Mark Segal, and Derek Gerstmann. A performance-oriented data parallel virtual machine for GPUs. In ACM SIGGRAPH 2006 Sketches, SIGGRAPH '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6. doi: http://doi.acm.org/10.1145/1179849.1180079. URL http://doi.acm.org/10.1145/1179849.1180079.

Simon L. Peyton Jones. The Implementation of Functional Programming Languages (Prentice-Hall International S Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. ISBN 013453333X.

Guy L. Steele, Jr. Rabbit: A Compiler for Scheme. Technical report, Cambridge, MA, USA, 1978.

Don Syme. The F# 2.0 Language Specification (April 2010), 2010. URL http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.pdf.

David Tarditi, Peter Lee, and Anurag Acharya. No Assembly Required: Compiling Standard ML to C. Technical report, ACM Letters on Programming Languages and Systems, 1990.

David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A Type-Directed Optimizing Compiler for ML. In IN ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, pages 181–192. ACM Press, 1995.

David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. SIGOPS Oper. Syst. Rev., 40:325–335, October 2006. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1168917.1168898. URL http://doi.acm.org/10.1145/1168917.1168898.

Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. J. Funct. Program., 8:367–412, July 1998. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796898003086. URL http://dx.doi.org/10.1017/S0956796898003086.

7 Abbildungsverzeichnis

8 Tabellenverzeichnis