Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 2477 Simulation Technology Nr. 49

Entwicklung eines Source-to-Source Compilers für explizite Optimierung

Jan Rapp

Studiengang:	Simulation Technology
Prüfer/in:	Jun.-Prof. Dr. rer. nat. Dirk Pflüger Prof. Dr. rer. nat. habil. Miriam Mehl
Betreuer/in:	Dipl.-Inf. Pfander, David
Beginn am:	30. November 2015
Beendet am:	30. Mai 2016
CR-Nummer:	D.3.4

Kurzfassung

Die Laufzeitoptimierung von Programmen involviert oft repetitive Transformationen des Source Codes, welche aufwendig sind und den Code schwerer verständlich machen. In dieser Arbeit wird ein Source to Source Compiler präsentiert, mit welchem derartige Transformationen automatisiert werden können. Dieser Compiler wurde für eine Teilmenge von OpenCL-C und C entwickelt und nutzt Pragmas, um anzugeben, wann welche Transformationen angewendet werden sollen. Die Transformationen können zudem noch parametrisiert sein. Durch selektives Anwenden der Transformationen und Durchprobieren mehrerer Parameterkombinationen können damit leicht viele unterschiedliche Versionen des Codes erstellt werden, die für Autotuning genutzt werden könnten. Bei Autotuning wird die Laufzeit der unterschiedlichen Versionen geschätzt und es kann die beste für die aktuelle Hardware ausgewählt werden.

Inhaltsverzeichnis

Die Literaturrecherche zu dieser Arbeit ist separat im Propaedeuticum.

Abbildungsverzeichnis

1. Einleitung

In dieser Arbeit sind mit Optimierungen immer Verringerungen der Laufzeit von einem Programm gemeint. Die Laufzeit von Software zu verringern, ist in vielen Fällen erwünscht, aber häufig mit großem Aufwand verbunden. Der Aufwand kommt oft daher, dass der Code geändert werden muss. Manche diese Änderungen sind generisch und nicht problemabhängig und müssen vom Programmierer häufig gemacht werden. Hier liegt ein gewisses Automatisierungspotential vor, welches in dieser Arbeit teilweise genutzt wird.

In dieser Arbeit bezeichnen wir Änderungen von Code als Transformation, wenn der Code nach der Änderung zumindest teilweise Ähnlichkeit zum Code vor der Änderung hat. Ziel ist es, Transformationen zu automatisieren, welche häufig angewendet werden und die Laufzeit verringern können. Das wird mit Hilfe von Compilerbautechniken angegangen. Genauer gesagt wird ein Source-to-Source Compiler präsentiert, welcher Transformationen in OpenCL-C Code automatisiert. OpenCL-C ist eine besonders für hoch parallele Rechnungen geeignete Programmiersprache, die auf unterschiedlichen Prozessoren, zum Beispiel auch GPUs, ausgeführt werden kann. OpenCL-C ist Teil von OpenCL [4] (Open Compute Language), was ein Standard über die Verwendung von Prozessoren, besonders bei Parallelrechnern, ist. Mit einem Source-to-Source Compiler ist ein Compiler gemeint, der Source Code als Ein- und als Ausgabe hat. Der in dieser Arbeit präsentierte Compiler hat OpenCL-C als Ein- und Ausgabe.

Die Transformationen werden von dem in dieser Arbeit präsentierten Compiler automatisch gemacht. Ob und wo die Transformationen angewandt werden, wird aber vom Programmierer vorgegeben. Er behält also die völlige Kontrolle.

In Kapitel 2 wird kurz die Programmierplattform OpenCL etwas näher erläutert, gefolgt von einer tieferen Einführung in den Compilerbau in Kapitel 3. Diese Vorgehensweisen und Mittel aus dem Bereich Compilerbau werden in den darauf folgenden Kapiteln 4 und 5 auf die konkrete Lösung des Problems angewendet. Kapitel 4 geht auf die konkrete Vorgehensweise der Source-to-Source Transformationen ein. Näheres zur Implementierung des Source-to-Source Compilers und wie er für Transformationen genutzt werden kann, wird in Kapitel 5 beschrieben. In Kapitel 6 werden drei Beispieltransformationen vorgestellt und es wird gezeigt, dass Laufzeitverbesserungen möglich sind. Das letzte Kapitel fasst die Arbeit kurz zusammen und gibt einen Ausblick auf mögliche weiterführende Transformationen.

2. OpenCL

OpenCL (Open Compute Language) ist eine Spezifikation für ein Framework, das parallele Berechnungen auf unterschiedlicher Hardware vereinfachen soll. Es eignet sich unter anderem für GPUs (Graphikkarten). Diese Hardware wird hier als Device bezeichnet. In der Tat gibt es für die großen Grafikkartenhersteller NVIDIA und AMD Implementierungen für die weit verbreiteten Betriebssysteme Windows, OS X und Linux. In dieser Arbeit ist mit OpenCL immer OpenCL in der Version 1.2 gemeint. Im Folgenden ist mit OpenCL auch die Implementierung gemeint.

Der Code, der auf dem Device ausgeführt wird, wird vom Hostprogramm gesteuert, ein Programm das ganz normal auf dem Computer ausgeführt wird. Das Hostprogramm gibt OpenCL den kompletten OpenCL-Code in Textform. Dieser wird dann von OpenCL für das Device compiliert und auf Anweisung vom Hostprogramm auf dem Device ausgeführt. Viele Möglichkeiten stehen zur Verfügung, den Speicher auf dem Device zu verwalten und Informationen mit Hostprogramm austauschen.

OpenCL ist speziell für Berechnungen ausgelegt, die sich gut parallelisieren lassen. Grob gesagt wird in OpenCL eine Berechnung in viele Teile aufgespalten, die sogenannten Workitems. Jedes Workitem führt den gleichen Code aus, hat aber einen eigenen Index.

OpenCL-C ist eine stark an C angelehnte Programmiersprache, in der die Programme geschrieben werden, die auf dem Device ausgeführt werden. Die Ähnlichkeit von OpenCL-C zu C ist sehr groß. Auf syntaktischer Ebene werden in OpenCL nur wenige Schlüsselwörter hinzugefügt. Es werden zum Beispiel weitere primitive Typen hinzugefügt, die sich syntaktisch genauso verhalten wie andere Typen in C.

3. Einführung in den Compilerbau

Um die Source-to-Source Transformationen durchzuführen, nehmen wir die gleiche Sichtweise auf das Programm wie ein normaler Compiler ein. Im folgenden werden einige Inhalte des Compilerbaus oberflächlich beschrieben, die für diese Arbeit relevant sind.

Compiler gibt es in vielen verschiedenen Varianten und sie sind häufig deutlich komplizierter als hier dargestellt. Oft haben sie auch mehr Funktionalität als einfach nur eine Quellsprache in eine Zielsprache zu übersetzen. Für detailliertere Informationen über Compilerbau kann zum Beispiel [5] herangezogen werden; dort wird eine ähnliche Sichtweise auf Compiler genommen wie in diesem Kapitel.

Compiler können als dreiteilig angesehen werden. Dann wird in Frontend, Middleend und Backend unterschieden. Der Source Code oder eine Repräsentation davon durchläuft die drei Teile in dieser Reihenfolge. Abbildung 3.1 gibt einen schematischen Überblick.

Das Frontend beschäftigt sich direkt mit der Quellsprache, der Programmiersprache in der das Programm geschrieben ist. Das Ergebnis des Frontends ist das Programm in einer Zwischenrepräsentation. Das Frontend ist meist zuständig für das Finden und Ausgeben von Fehlern im Programm. Im Middleend und im Backend muss man sich dann (abgesehen von Bugs im Frontend) nur noch mit korrekten Programmen auseinandersetzen.

Das Middleend transformiert das Programm in der Zwischenrepräsentation und versucht Code zu optimieren. Zum Schluss generiert das Backend aus der Zwischenrepräsentation die Zielsprache, zum Beispiel x86 Maschinencode.

Zusätzlich kann die Zwischenrepräsentation quell- und zielsprachenunabhängig sein. Dann können für unterschiedliche Quellsprachen Frontends und für unterschiedliche Zielsprachen Backends entwickelt werden, die alle dieselbe Zwischenrepräsentation erzeugen oder einlesen. Damit können dann alle Quellsprachen in alle Zielsprachen compiliert werden und das Middleend kann für Optimierungen dazwischen geschaltet werden.

Der Rest des Kapitels beschäftigt sich mit Teilen des Frontends, welche für dieses Projekt besonders interessant sind.

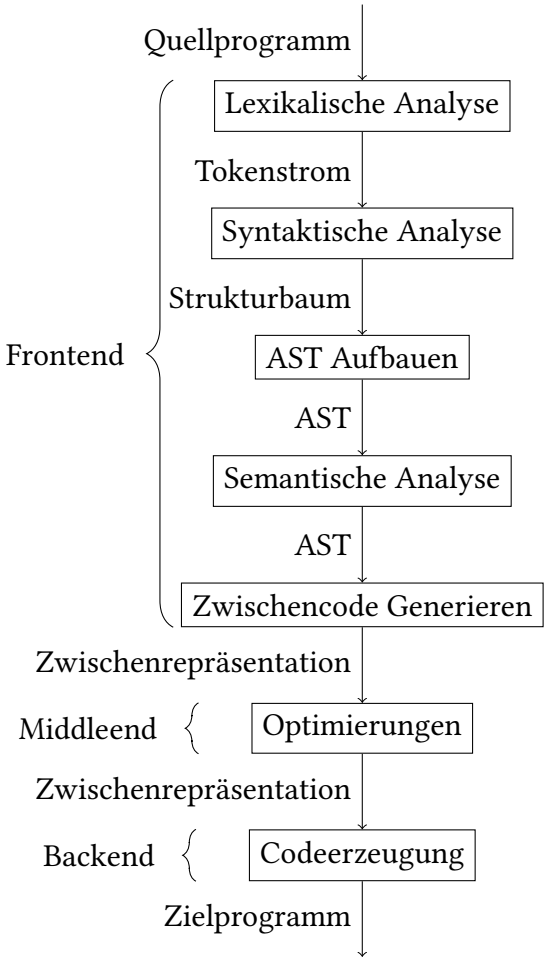


Abbildung 3.1.: Zu sehen ist, wie der Programmcode die unterschiedlichen Teile eines Compiler durchläuft und welche Repräsentation er annimmt.

3.1. Lexikalische Analyse

Kurz gesagt werden die Zeichen des Eingabetextes in der lexikalischen Analyse in die Symbole der Sprache zusammengefasst. Beispiele für ein Symbol in C sind: eine schließende geschweifte Klammer `}`; ein Operator wie ein Plus `+` wenn kein weiters Plus folgt (ansonsten wäre es der Plus Plus `++` Operator), ein ganzes Schüsselwort wie `for` oder ein Variablenname wie `ein_langer_variablenname`.

Der erste Teil in der lexikalischen Analyse wird das Tokenizing genannt. Beim Tokenizing wird entschieden, wo der Eingabetext unterteilt wird. In C ist das immer bei einem Whitespace, vor und nach Klammern, zwischen einem Operator- und einem Nicht-Operatorzeichen und nach etwas komplizierteren Regeln zwischen zwei Operatorzeichen.

Diese Token werden dann kategorisiert. Bei den Kategorien handelt es sich um die Terminalsymbole, welche später in der syntaktischen Analyse weiterverarbeitet werden. Oft reicht es, nur die Kategorie des Tokens zu speichern (wie zum Beispiel bei einer öffnenden Klammer). Manchmal sind aber noch andere Informationen über den Token relevant, die auch gespeichert werden müssen. Zum Beispiel haben alle Identifier (Bezeichner) die gleiche Kategorie und erst nach der syntaktischen Analyse werden die eigentlichen Identifier wieder interessant.

3.2. Syntaktische Analyse

Aus der lexikalischen Analyse haben wir nun eine Folge von Terminalsymbolen, die wir weiterverarbeiten wollen. Fasst man die Terminalsymbole einfach nur als Folge von Symbolen auf, dann ist die syntaktische Analyse ein Immer-wieder-Gruppieren von Symbol-Teilfolgen zu neuen Symbolen (so genannten Nichtterminalsymbolen), bis nur ein Symbol übrig bleibt. Diese letzte Symbol muss als ein spezielles Symbol, das Startsymbol, designiert sein, damit die syntaktische Analyse erfolgreich ist. Falls es nicht möglich ist, die Folge an Terminalsymbolen mit erlaubten Gruppierungen bis zum Startsymbol hin zusammenzufassen, schlägt die syntaktische Analyse fehl. Welche Gruppierungen erlaubt sind, sagt die Grammatik[1] aus. Eine Grammatik besteht aus Produktionen, die aussagen, welche Folgen an Symbolen zu welchem Nichtterminalsymbol zusammengefasst werden dürfen. Damit entsteht ein Strukturbaum, in dem jedes Nichtterminalsymbol ein Knoten ist, dessen Kinder die Symbole sind, aus denen es zusammengesetzt wurde. Die Blätter des Strukturbaumes sind Terminalsymbole und die Wurzel ist das Startsymbol.

In Abbildung 3.2 wird beispielhaft eine Grammatik gezeigt und eine syntaktische Analyse vorgenommen. Außerdem wird gezeigt, wie der Strukturbaum wenig geeignet dazu sein kann, den Code zu verstehen und zu transformieren. Dieses Problem wird im nächsten Kapitel angegangen.

3.3. AST

Ein AST (Abstract Syntax Tree) ist wie der Name andeutet, eine Baumdarstellung der Syntax, welche zu einem gewissen Grad abstrahiert ist. Es handelt sich grob um einen vereinfachten Strukturbaum, der aber immer noch fast die komplette Syntax repräsentiert. Das hilft um den Umgang damit zu vereinfachen; Klammern zum Beispiel können oft weggelassen werden, denn Gruppierungsinformationen ergeben oft aus der Baumstruktur. Waren die Klammern unnötig, können sie aus dem AST dann nicht wieder hergestellt werden. Siehe Abbildung 3.3 für ein Beispiel.

[1] In dieser Arbeit sind Grammatiken eindeutig und kontextfrei. Dies entspricht also nur einer Untermenge von dem, was sonst unter Grammatiken verstanden wird.

```
expression : expression "+" term
           | expression "-" term
           | term
           ;
term : term "*" unary
     | term "/" unary
     | term "%" unary
     | unary
     ;
unary : "-" atom
      | "+" atom
      | atom
      ;
atom : "(" expression ")"
     | integer_literal
     | float_literal
     ;
```

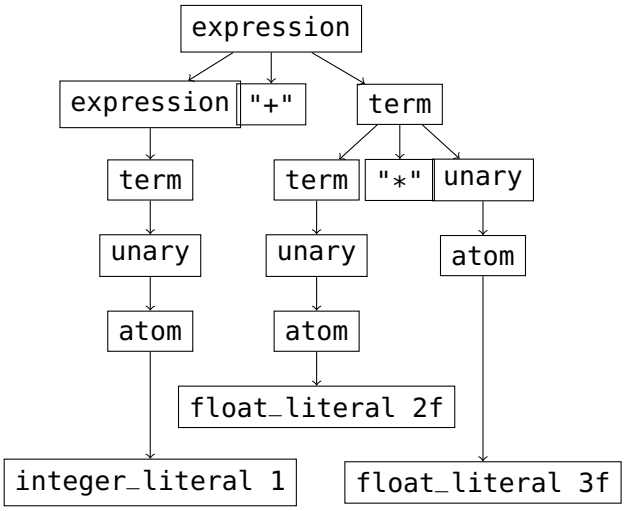
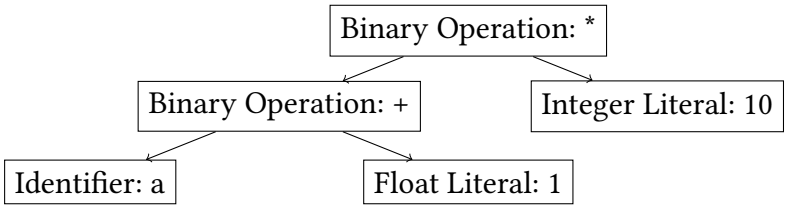


Abbildung 3.2.: (Links) Eine kleine Beispielgrammatik für arithmetische Ausdrücke. Die Symbole `integer_literal` und `float_literal`, sowie alles in Anführungszeichen sind Terminalsymbole. Die Grammatik stellt sicher, dass Operationen mit höherer Präzedenz im resultierenden Strukturbaum zusammen gehören. Operatorpräzedenz ist also schon Teil der Syntax und nicht erst der Semantik. Außerdem werden Klammern und das unäre Minus richtig gruppiert. Diese Grammatik besteht aus Regeln die mit Semikola beendet werden. Jede Regel beginnt mit dem Nichtterminalsymbol, welches durch die Regel definiert wird. Danach folgt ein Doppelpunkt und eine oder mehrere, durch vertikale Striche getrennte, Produktionen für dieses Nichtterminalsymbol. (Rechts) Der Strukturbaum des Ausdrucks `1 + 2f * 3f` auf den die syntaktische Analyse mit der Grammatik links angewendet wurde. Die Terminalsymbole sind die drei Zahlen und die beiden Operatoren. Das nachstehende `f` an den Zahlen bedeutet, dass es sich um **`float`**s handelt. Das Startsymbol muss in diesem Fall `expression` sein. Wie hier zu sehen ist, ist der resultierende Strukturbaum unnötig groß, zusammenhängende Teile sind weit auseinander. Wie während der Bearbeitung dieser Arbeit festgestellt wurde, macht das die Verarbeitung schwer.




```
Binary Operation: *
  Binary Operation: +
    Identifier: a
    Float Literal: 1
  Integer Literal: 10
```

Abbildung 3.3.: AST (Abstract Syntax Tree) der zum Ausdruck `((a + 1.00f)* 10)` gehören könnte. Links als Graph und Rechts als Text, wobei die Hierarchie durch die Einrückung ausgedrückt wird. Man beachte, dass Klammern weggelassen wurden und dass die genaue Darstellung der Gleitkommazahl vernachlässigt wurde. Dass die Additionsoperanden zusammengehören, ist aber noch ersichtlich. Mit der Grammatik in Abbildung 3.2 könnte der Strukturbaum aus dem Ausdruck `((a + 1.00f)* 10)` erzeugt worden sein. Dieser wäre aber komplizierter als der Strukturbaum von `1 + 2 * 3` aus Abbilding 3.2 denn der Teil in der Klammer ist eine `expression`, die mehrere Knoten braucht, bis sie in die Multiplikation eingebunden werden kann.

Im Gegensatz zum Strukturbaum ist der AST deutlich einfacher und kann auch anders aufgebaut sein. In der C Grammatik zum Beispiel gibt es 370 unterschiedliche Regeln. In einem Strukturbaum sind das 370 unterschiedliche Knotentypen. Im Vergleich dazu hat der C AST der in dieser Arbeit implementiert wurde nur 43 Knotentypen. Diese Diskrepanz kommt zwar auch daher, dass der AST nicht alle C Funktionalität unterstützt (genauer 70 Regeln) aber es kommt auch daher, dass ein AST deutlich einfacher sein kann. Die Grammatik muss eben aus den Token einen Strukturbaum erstellen können (und sollte dabei auch nicht mehrdeutig sein) und muss von dem Parser verarbeitet werden können. Der AST muss die Syntax nur darstellen können. Ein Parser ist ein Algorithmus (oder dessen Implementierung), welcher die syntaktische Analyse bewerkstelligen kann.

3.4. Semantische Analyse

Viele Programmiersprachen haben mehr Anforderungen an den Programmcode als durch eine syntaktische Analyse überprüft werden kann. Die ausstehenden Überprüfungen und das Sammeln von weiteren Informationen, die später benötigt werden, kann man als semantische Analyse bezeichnen. In statisch typisierten Programmiersprachen werden hier zum Beispiel alle Anforderungen an Typenkonsistenz getestet.

4. Vorhaben

Die Aufgabenstellung dieser Arbeit ist explizite Transformationen in Programmcode zu vereinfachen, indem Compilerbautechniken angewandt werden. Dieses Problem wird für die Programmiersprachen C und OpenCL-C angegangen, wobei der Fokus auf OpenCL-C liegt. Die Software, die in dieser Arbeit entwickelt wurde, wurde ALR[1] genannt.

Die Compilerbaugrundlagen aus dem letzten Kapitel werden nun eingesetzt, um einen gegebenes Programm zu verstehen. Genauer wird in ALR nun aus diesem Programm ein AST erzeugt, der das Programm repräsentiert. Die expliziten Transformationen werden in diesem AST vorgenommen und danach wird aus dem AST wieder ein normales textuelles Programm erzeugt, das die Transformationen beinhaltet und ausgeführt werden kann.

Um ALR mitzuteilen welche Transformationen es für ein gegebenes Programm anwenden soll, werden Annotationen direkt in das Programm eingefügt. In OpenCL-C und C lässt sich das durch Pragmas machen. Pragmas sind für derartige Annotationen gut geeignet, denn sie sollen ignoriert werden wenn die Software, welche den Code verarbeitet, dieses Pragma nicht erkennt (Kapitel 6.10.6 [1] und Kapitel 6.10 [4]). Die Position des Pragmas im Programmtext ist ein wichtiges Mittel, um anzugeben, wo die Transformation angewendet werden soll. Konkret sieht das zum Beispiel wie in Abbildung 4.1 aus.

4.1. Einbindung in OpenCL

OpenCLs interface ist eine C-API, die genutzt wird, um mit OpenCL Code auf dem Device auszuführen. Dafür muss der Code in Textform an die entsprechende Funktion übergeben werden, die diesen für ein bestimmtes Device compiliert. Diese Funktion hat als Rückgabe ein Objekt, das dazu genutzt werden kann, den gerade compilierten Code auszuführen. Solange der Code noch in Textform ist, kann er nun von ALR transformiert werden. Der Pseudocode in Abbildung 4.2 zeigt grob, wie das aussieht und dass es dadurch, dass der Code sowieso als string durch das Programm gereicht wird, unproblematisch ist, die zusätzliche Transformation zwischenzuschieben.

[1] Der Name hat keine weitere Bedeutung.

my_OpenCL_code.cl:

```
1  kernel void main(global const double* data1, global const double* data2,
2     uint size, global double* result) {
3    int globalIdx = get_global_id(0);
4    int localIdx = get_local_id(0);
5    double accumulator = 0;
6
7    #pragma alr(1) localcache(64,localIdx,data1 data2)
8    for (int i = 0; i < size; ++i) {
9
10     #pragma alr(2) unroll(10)
11     for (int j = 0; j < 10; ++j) {
12       accumulator += do_calculation(data1[j * size + i], data2[j * size + i],
13                                globalIdx);
14     }
15   }
16   result[globalIdx] = accumulator;
17 }
```

Abbildung 4.1.: In diesem Beispiel wird zuerst die `localcache` Transformation ausgeführt (Annotation in Zeile 7), und dann die `unroll` Transformation (Annotation in Zeile 10.

```
1 source_code = read_file("mein_OpenCL_code.cl")
2 source_code = alr::transform(source_code, {1,2}) // <== HIER
3 programm = compile_OpenCL_code(source_code)
4 programm.execute(args...) // wird auf Device ausgeführt
```

Abbildung 4.2.: Pseudocode, welcher die für diese Arbeit relevanten Aspekte von OpenCL demonstriert. In der Zeile die mit `<== HIER` markiert ist, werden die Transformationen angewandt. Diese Zeile kann komplett weggelassen werden und die Transformationen werden dann einfach nicht angewandt. Die Funktion `alr::transform` existiert aber wirklich so und der zweite Parameter davon wird in Kapitel 4.2 beschrieben.

4.2. Reihenfolge und Auswahl von Transformationen

Die Reihenfolge, in der die expliziten Transformationen angewendet werden, muss auch explizit spezifiziert werden. Alle Pragmas beginnen mit `alr(#)` wobei # irgend ine ganze positive Zahl ist. Damit ist diese Zahl mit der Transformation, die in dem Pragma beschrieben ist, assoziert. Wenn ALR den Code transformiert, muss eine Liste an Zahlen angegeben werden und ALR führt die mit den Zahlen assoziierten Transformationen in der Reihenfolge aus, wie sie in der Liste angegeben sind. Damit lässt sich auch leicht eine Transformation ein- oder ausschalten, die assozierte Zahl wird einfach nicht übergeben.

5. Elemente des Aufbaus und Verwendungsmöglichkeiten von ALR

ALR ist in C++ 11 implementiert und nutzt keine zusätzliche Bibliotheken. Es werden aber Flex und Bison (dazu gleich mehr) genutzt, um die Teile der lexikalischen und syntaktischen

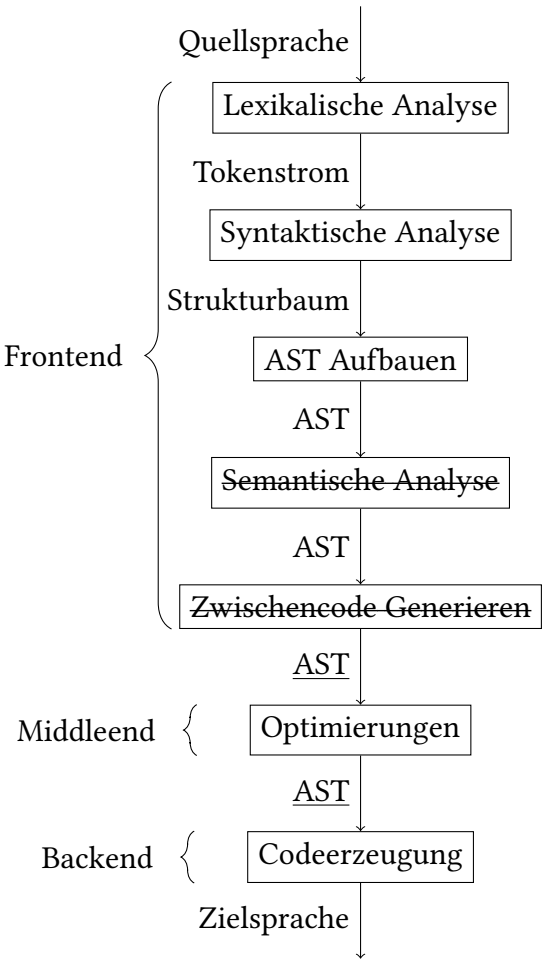


Abbildung 5.1.: Veränderungen in dem Compileraufbau von ALR im Vergleich zu Abbildung 3.1.

Analyse zu generieren. Es handelt sich großteils um einen C Parser, welcher zu einem OpenCL-C erweitert wurde. Wie viel der C und OpenCL-C Syntax ALR versehen kann wird im folgenden Unterkapitel erläutert.

Wie in Abbildung 5.1 dargestellt wird weist ALR einige Unterschiede zu einem typischen Compiler auf. Die semantische Analyse wird im Frontend komplett ausgelassen, was die Komplexität, im Gegensatz zu einem typischen, Kompiler deutlich verringert. Zudem wird als Zwischenrepräsentation der AST gewählt.

Die expliziten Optimierungen werden alle auf dem AST ausgeführt und auf welche Weisen das passiert wird gegen Ende dieses Kapitels beschrieben.

5.1. Kompatibilität

Um die Implementierung zu vereinfachen wurden viele Teile der Programmiersprachen nicht berücksichtigt. Die Hoffnung ist, dass die wichtigste Funktionalität noch zur Verfügung steht und das portieren von anderem Code nicht unnötig eingeschränkt wird.

5.1.1. C

Einige syntaktische C11 Features wurden nicht implementiert:

1. Jede Form von `typedefs` oder `enmums`. Das Benutzen der Enumkonstanten (die Elemente des Enums) geht aber von syntaktische Seite her immer. Enumkonstanten dürfen nur an Stellen genutzt werden, an denen auch eine Variable stehen könnte. Dann wird die Enumkonstante einfach als Variable angesehen und führt zu keinen weiteren Komplikationen. Enumkonstanten sind aber eigentlich Konstanten und bei einer semantischen Analyse würde es dann an den Stellen zu Komplikationen kommen, an denen eine Konstante gefordert wird und eine Enumkonstante angegeben ist; denn die Enumkonstante wurde von ALR als Variable angesehen und wäre damit nicht konstant. Eine Konstante wie `CLK_LOCAL_MEM_FENCE` ist davon auch betroffen und die brauchen wir in Kapitel 6.3.
2. Pragmas sind nur an Stellen erlaubt an denen auch ein Statement oder eine globale Definition stehen darf.
3. Declaration von Unions.
4. Typspezifizierer die andere Spezifizierer oder Qualifizierer nach sich haben. Typspezifizierer (in C type-specifier; wie **`int`**, **`void`**, etc.) müssen alle anderen Spezifizierer oder

Qualifizierer (wie zum Beispiel Typqualifizierer (zum Beispiel **`const`**) oder Funktionenspezifizierer (zum Beispiel **`inline`**)) vor sich haben. In C ist eigentlich **`const int`** und **`int const`** das gleiche; ALR aber erlaubt nur die erste Variante.[1]

5. Mehr als ein Typspezifizierer hintereinander. So etwas wie **`long long`** oder auch **`unsigned long`** ist in ALR nicht erlaubt.[2]
6. Alles was mit `static_asserts` zusammenhängt.
7. Bit Fields in **`struct`** Deklarationen (zum Beispiel `struct { uchar i:1; uchar j:1; uchar k:6; }`).
8. Type Qualifier (wie `const` etc.), `static` oder `*` (von Variablen Length Arrays) in den Eckigen Klammern bei Array Declarationen.
9. Funktionen Parameter ohne Typen. Vor dem ANSI C Standard hatten Funktionsparameter keine Typen. Dieses C wird als K&R C bezeichnet, und wird in dem Buch „The C Programming Language“ von Kernighan und Ritchie [6] beschrieben, das für das `Hello World!` Programm bekannt ist. Aus Kompatibilitätsgründen ist das teilweise noch erlaubt.
10. Variable Parameter Anzahl (zum Beispiel `printf(format, ...)` [3]).
11. Designatorsyntax für Initializerlisten (zum Beispiel `{.i = 5, .j[2] = 3}`).
12. For Schleifen die keine Deklaration sondern einen Ausdruck vor dem ersten Semicolon haben. Davon ausgenommen sind aber Schleifen die nichts vor dem ersten Semikolon haben.
13. Schleifen die nichts zwischen dem ersten und zweiten Semikolon haben. In C wird das gleich behandelt wie wenn dort eine wahre Aussage steht.
14. Funktionen die Variablen außerhalb des Funktionenkörpers deklarieren (zum Beispiel: `int main() int i; char* j; { ... }`).
15. C Features, welche eines der Folgenden Keywords verwenden: `_Alignas`, `_Alignof`, `_Atomic`, `_Complex`, `_Generic`, `_Imaginary`, `_Thread_local` und `__func__`.

[1] Möglicherweise wäre es konsistenter Qualifizierer und Spezifizierer nur *nach* dem Typspezifizierer zu Erlauben (wenn man nur danach oder nur davor erlaubt). Das wäre dann wie bei Pointertypen, bei denen die Qualifizierer immer nach dem `*` stehen müssen und passt auch dazu, dass man Typen in C von „Innen nach Außen“ lesen muss. Diese Variante scheint allerdings weniger gebräuchlich zu sein.

[2] In OpenCL gibt es aber für alle Typen ohne Vorzeichen die alternative mit einem vorgestellten `u`; zum Beispiel also `ulong`.

[3] Das wäre Problematisch wenn man semantische Analysen anfangen würde und C code unterstützen will. Dann muss man ja irgendwie die Declarationen der Standardbibliothek einbinden (**`#include`**) und parsen können und selbst in der ältesten Standardbibliothek von C kommt ein `printf` vor.

Der AST in ALR ist zudem nicht in der Lage mehrere Declarationen pro Statement zu haben. ALR kann solche zwar parsen, aber in der AST-Darstellung werden diese in mehrere Statements aufgespaltet.

5.1.2. OpenCL-C

ALR befasst sich mit OpenCL in der Version 1.2 [4]. Folgende Anpassungen wurde gemacht damit ALR nicht nur mit C Code sondern auch mit OpenCL Code umgehen kann:

1. Hinzufügen der folgenden Keywords an den entsprechenden Stellen `kernel`, `global`, `local`, `constatnt`, **`private`**, `read_only`, `write_only`, `read_write`, `__kernel`, `__global`, `__local`, `__constatnt`, `__private`, `__read_only`, `__write_only` und `__read_write`. Die in Kapitel 6.1.9 der OpenCL Spezifikation [4] aufgelistet werden.
2. Hinzufügen der Typen `uchar`, `ushort`, `uint`, `ulong`, **`bool`**, `half`, `size_t`, `ptrdiff_t`, `intptr_t`, `uintptr_t`, `image2d_t`, `image3d_t`, `image2d_array_t`, `image1d_t`, `image1d_buffer_t`, `image1d_array_t`, `sampler_t` und `event_t` aus Kapitel 6.1.1 von [4].
3. Hinzufügen von den Vectortypen `charN`, `ucharN`, `shortN`, `ushortN`, `intN`, `uintN`, `longN`, `ulongN`, `floatN` und `doubleN` wobei `N` 2,4,8, oder 16 sein darf. Beschrieben in Kapitel 6.1.2 von [4].
4. `__attribute__(...)` Annotationen an Funktionen mit der Einschränkung, dass alle Zeichen innerhalb der Klammern korrekt geklammert sein müssen. Also `__attribute__(hallo("(")` ist nicht möglich, obwohl die öffnende Klammer innerhalb eines Strings ist. Der Attributsyntax ist nicht Standard in C wird aber in OpenCL (in Kapitel 6.11 von [4]) standardisiert und ist von GCC übernommen.

Viel Funktionalität, die OpenCL-C hinzufügt, muss in der syntaktischen Analyse nicht betrachtet werden, weil diese erst in der semantischen Analyse betrachtet werden würde. Darunter fallen zum Beispiel Funktionen oder Konstanten.

OpenCL Vector Literale

OpenCL führt neue Literale für Vectoren ein ([4] Kapitel 6.1.6). Zum Beispiel kann ein `float4` mit `(float4)(4.0f,3.0f,2.0f,1.0f)` erstellt werden. Das Problem ist aber, dass der Syntax mit einer Kombination aus Konvertierungs- und Kommaoperator kollidiert. Nach Kapitel 6.3.l. der OpenCL Specification [4] ist der Kommaoperator (`,`) so definiert das die Argumente von links nach rechts evaluiert werden und das Resultat vom Rechtesten zurückgegeben wird. Anstatt `(float4)(4.0f,3.0f,2.0f,1.0f)` als Vector Literal zu sehen kann man die zweite Klammer auch einfach als Ausdruck von **`float`** sehen dessen Resultat zu einem `float4` konvertiert wird. Im Kapitel 6.2.2 „Explicit Casts“ wird als Beispiel für ein Cast sogar

```
1 float f = 1.0f;
2 float4 va = (float4)f;
3 // va is a float4 vector with elements (f, f, f, f).
```

aufgeführt. Eine Abänderung zu

```
1 float f = (4.0f,3.0f,2.0f,1.0f)
2 float4 va = (float4)f;
```

hat dabei natürlich keinen Effekt. Eine Abänderung zu

```
1 float4 va = (float4)(4.0f,3.0f,2.0f,1.0f);
```

verändert aber den Wert von `va` zu einem Vector mit vier unterschiedlichen Elementen. Hier gehe ich davon aus, dass es als Vector Literal interpretiert wird obwohl das nicht klar ist[4].

Vector Literale wurden in ALR nicht implementiert.

5.2. Lexikalische und Syntaktische Analyse

In ALR werden zum lexen und parsen jeweils die GNU Programme Flex und Bison genutzt um den Lexer und Parser für ALR zu generieren. Für diese Programme hat schon jemand aus dem C Standard die nötigen Definitionen zusammen geschrieben und entsprechd in Flex und Binson Format gebracht [2, 3].

Sowohl Flex und Bison generieren C oder C++ Code, der dann den Lexer oder Parser darstellt. Dieser Code wird bei Flex aus regulären Ausdrücken generiert, welche die Token definieren. Bei Bison wird der Code aus einer Grammatik generiert. Schon die Abbildung 3.2 aus dem Kapitel 3 zeigt eine Grammatik wie sie von Binson genutzt werden kann.

5.3. AST in ALR

Um mit ALR Transformationen zu implementieren muss man sich hauptsächlich mit dem AST auseinandersetzen. Der AST in ALR ist eine Baumdatenstruktur, in der Knoten Zeiger auf ihre Kinder haben. Jeder Knoten hat einen Knotentyp, welcher bestimmt was der Knoten darstellt und welche Kinder er hat. Abbildung 5.2 listed eine Auswahl der Knotentypen, wie sie in ALR vorkommen, auf. Mit diesen Knoten kann man einen großen Teil der häufig benutzen Programmierelemente der C Programmiersprache darstellen. Auch viele Erweiterungen die durch OpenCL dazukommen werden davon einfach unterstützt ohne das der AST angepasst werden muss.

[4] Es gibt aber in Kapitel 6.1.6 in [4] über Vector Literale das Beispiel `float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);` und Aufgrund des Kapitels in dem es aufaucht, gehe ich davon aus, dass es sich auch um ein Vector Literal handelt, so wie es dort beschrieben ist.

```
IntLit:Ex        | v^str                              // whole number literal

IdEx:Ex          | id^str                             // identifier expression
ArrEx:Ex         | ex:Ex idx:Ex                       // array indexing
BinEx:Ex         | op^str left:Ex right:Ex            // binary operation

BlkSt:St         | sts@St                             // block statement
ForSt:St         | init@DeclB cond:Ex inc?:Ex st:St // for statement

Decl:DeclB       | ty:Ty name^str init?:Ex            // declaration
Func:TItem       | ty:Ty name^str body:St             // function definition
```

Abbildung 5.2.: Wichtige Knotentypen des AST in ALR. Eine Vollständige Liste der 43 Knotentypen ist im Anghang B. Jede Zeile beschreibt einen Knotentyp: Der Aufbau ist `<Knotentyp>:<Basistype> | <ein oder mehrere mit Whitespace getrennte Eigenschaften oder Kindknoten>`. Die Eigenschaften oder Kindknoten bestehen aus Name, Art und Typ in der Anordnung `<Name><Art><Typ>`. Ist die Art ein Dach (`^`) handelt es sich um eine Eigenschaft, welche keine weiteren Kindknoten sondern andere weitere Informationen über den Knoten enthält. Der Typ kann bei einer Eigenschaft jeder C++ Typ sein (`str` ist eine Abkoerzung fuer `std::string`) Die anderen Arten sind Kindknoten und der Typ muss auch von `Nd` abgeleitet sein. Ein Kindknoten der Art Doppelpunkt (`:` oder `?:`) ist ein einzelner Kindknoten vom entsprechenden Typ, wobei dieser bei der zweiten Variante optional ist. Die Art At `@` ist eine Liste von Kindknoten wobei jedes Element den genannten Typ hat.

```
St:Nd     // statement
Ex:Nd     // expressions
Ty:Nd     // type
TItem:Nd  // translation item
DeclB:Nd  // declaration base
TUnitB:Nd // translation unit base
```

Abbildung 5.3.: Die Basistypen für AST Knoten. Allesamt haben die abstrakte Klasse `Nd` (Node) als Basis.

Jeder Knotentyp hat einen Basistyp (siehe Abbildung 5.3) mit dem die Knoten gruppiert werden.

Implementierung

Die AST Knotentypen und deren Basistypen werden von genau dem Text generiert der in Abbildung 5.2 und 5.3 zu sehen ist. Dieser Text wird eingelesen und geparst. Damit werden für alle Typen Klassen erstellt, welche die spezifizierten Felder mit den richtigen Typen haben. Es werden auch Überprüfungen eingebaut damit Felder die nicht optional sind auch nicht null sind. Die eigentlichen Knoten sind Shared Pointer. Zum Beispiel ist `ForSt` eine using-Directive auf einen Shared Pointer, also `using ForSt = std::shared_ptr<ForSt_>;` und `ForSt_` ist die unterliegende Klasse mit den Feldern und Methoden. Die Knoten sind Pointer damit sie wiederverwendet werden können, wie es im nächsten Kapitel beschrieben wird.

5.3.1. AST immutability

Im AST haben Knoten Zeiger auf ihre Kindknoten. Kindknoten haben aber keine Zeiger auf die Elternknoten, da, wie wir gleich sehen werden, diese nicht eindeutig sind. Jeder Knoten ist nämlich immutable (unveränderlich) und um Änderungen vorzunehmen muss der Knoten neu erstellt werden. Damit der komplette AST an einer Stelle geändert werden kann müssen alle Knoten von dieser Stelle bis zur Wurzel hin ausgetauscht werden. Jeder dieser Knoten ändert sich, denn in jedem wird ein Kindknoten ausgetauscht. Da Kindknoten ihren Elternknoten nicht kennen, können sämtliche andere Knoten (und deren Teilbäume) aber einfach weiter genutzt werden und der Aufwand hält sich damit in Grenzen (siehe Abbildung 5.4 für ein Beispiel).

Die AST Knoten haben die Basistypen aus Abbildung 5.3. Sie sind von Node (`Nd`) abgeleitet, um eine gemeinsame Basis zu schaffen. Die Klassen, welche die Basistypen repräsentieren, werden nie instantiiert und dienen ausschließlich als Basis für andere Knoten. Jeder Knotentyp muss von solch einem Basistype abgeleitet sein. Die einzige erlaubte Hierarchie für Knotentypen ist also `Nd <: Basistype <: Knotentype`, was vieles vereinfacht. Alle Typen von Properties die Kindknoten enthalten müssen ein Basistyp sein.

Meistens sind Knoten mit dem selben Basistype austauschbar. Am wichtigsten ist es wohl zu sagen, dass eine Funktion ein Statement als Kindknoten hat, welches aber ein Blockstatement sein muss [5].Auch sonst schützen die Typen nicht vollständig gegen inkorrekte Nutzung, z.B müssen viele Strings korrekt sein, ein Identifer darf zum Beispiel keine Leerzeichen enthalten.

[5]Das ist etwas unschön aber in der Praxis scheint es kein Problem zu sein, da die Notwendigkeit Statements anzuschauen, die nicht schon in einem Block sind einfach nicht gegeben zu sein scheint. Damit passiert es auch nicht den Körper einer Funktion aus versehen durch ein Statement auszutauschen, dass kein Block ist.

Man könnte dieses Problem auch entfernen, indem man eine ähnliche Methode anwendet wie bei Declarationen. Siehe Anhang B. Man fügt einen neuen Knotentyp `Blk` ein, der eine Liste Statements hält und nutzt ihn an den zwei Stellen an denen man diesen braucht. Konkret könnte es mit folgenden Änderungen zu den Definitionen in Abbildung 5.2 gelöst werden:

```
BlkB:Nd
Blk:BlkB |sts@St
```

```
TUnit:
  Func: main
    FuncTy:
      QualTy: [kernel]
        IdTy: int
    BlkSt:
      /* Rest Inhalt der Funktion */
      ForSt:
        Decl: i
          IdTy: int
          IntLit: 0
        BinEx: <
          IdEx: i
          IntLit: 1024     -- Änderung Hier -->
        PreEx: ++
          IdEx: i
        BlkSt:
          /* Inhalt dieser for-Schleife */
      /* Rest Inhalt der Funktion */
    /* Restiliche Funktionen */
```

```
TUnit:
  Func: main
    FuncTy:
      QualTy: [kernel]
        IdTy: int
    BlkSt:
      /* Unverändert */
      ForSt:
        Decl: i
          IdTy: int
          IntLit: 0
        BinEx: <
          IdEx: i
          IntLit: 2048
        PreEx: ++
          IdEx: i
        BlkSt:
          /* Unverändert */
      /* Unverändert */
    /* Unverändert */
```

Abbildung 5.4.: Der AST vor (links) und nach (rechts) einer Änderung der Schleifenobergrenze. Dadurch, dass jeder Knoten unveränderlich ist müssen auch transitiv alle Elternknoten von dem veränderten Knoten neuerstellt werden damit der jeweilige Kindknoten getauscht werden kann. Die neuen Knoten sind fett geschrieben.

Für die Knotentypen `Decl` und `TUnit` gibt es jeweils einen eigenen Basistyp `DeclB` und `TUnitB`. Ein Objekt des entsprechenden Basistyps kann also, falls er nicht `nullptr` ist, immer zu dem entsprechenden Knotentyp gecastet werden.

5.4. Umgang mit dem AST in ALR

5.4.1. Transformer zur AST Änderung

Ein Knoten, von dem nicht alle Elternknoten bekannt sind, kann weder identifiziert noch geändert werden. Eine einfache aber recht limitierte Variante Änderungen vorzunehmen stellt die Transformerklasse zur Verfügung. Hier wird der gesamte Baum traversiert und für jeden AST Knotentyp wird eine bestimme Methode aufgerufen (wie bei einem Visitor Pattern). Diese

```
BlkSt:St blk@BlkB
Func:TItem | ty:Ty name^str body:Blk
```

Alternativ könnte man einfach die aktuelle `Func` in `Func:TItem | ty:Ty name^str body@St` ändern. Das wäre nicht so gut, da aufeinanderfolgende Statements besonders interessant sind und man nach ihnen nicht an zwei möglichen Knotentypen suchen will.

```
1  using namespace alr::ast;
2  struct ReplaceIdExTransformer : public Transformer {
3    std::string id;
4    Ex replaceEx;
5
6    Ex tIdEx (IdEx idEx) override {
7      return idEx->id == id ? replaceEx : idEx;
8    }
9  };
10
11 template<class T>
12 typename T::element_type::kind_type ReplaceIdEx(T input, std::string id, Ex
       replaceEx) {
13   ReplaceIdExTransformer transformer(id, replaceEx);
14   return cast<typename T::element_type::kind_type>(
15       input->transformNd(transformer)
16     );
17 }
```

Abbildung 5.5.: Ersetzen aller Identifier in dem Teilbaum `input`, welche einen bestimmten Namen haben. Normalerweise ist das Resultat eines Transformators irgend ein Knoten mit dem selben Basisknotentyp wie die Eingabe. Durch diese Transformation kann sich aber kein Knotentyp ändern, deshalb können wir in Zeile 14 wieder zu dem Knotentypen der Eingabe casten.

Methoden können nun einen veränderten Knoten zurückgeben und damit einen neuen AST aufbauen in dem nur die Knoten geändert sind, die auch geändert werden müssen. Um die Transformerklasse nutzen zu können, muss eine abgeleitete Klasse erstellt werden in welcher die gewünschten Methoden überschrieben werden. Die ursprünglichen Implementierungen kümmern sich um das korrekte Traversieren und die korrekte partielle Neuerstellung des Baumes. Manchmal lohnt es sich diese Implementierungen als Ausgang für die überschriebenen Methoden zu nutzen. Ein sehr einfaches Beispiel ist in Abbildung 5.5 zu sehen.

5.4.2. VisitorReplace zur AST Änderung

Der VisitorReplace ist, ähnlich wie der Transformer, ein Visitorpattern. Es gibt also eine Methode pro Knotentyp, die aufgerufen wird wenn wenn ein Knoten mit diesem Knotentyp betrachtet wird. Dabei wird zusätzlich zu dem Knoten auch noch ein Replacerobjekt mitgegeben, mit dem man (auch nach dem dieser Methodenaufruf vorbei ist) den Knoten ersetzen kann. Ist der Knoten Teil einer Knotenliste, kann auch davor und danach eingefügt werden. Falls der Knoten in einer Knotenliste oder optional ist, kann er entfernt werden.

Bei einem Durchlauf von einem VisitorReplace können nun die Replacerobjekte an den relevanten Stellen gesammelt werden und auch sonst relevante Informationen gespeichert werden. Danach können nun die Replacerobjekte genutzt werden, um den AST zu modifizieren. Bei jeder Modifikation wird der AST von der Stelle der Änderung bis zur Wurzel ausgetauscht und die neue Wurzel weiterverwendet. Die Replacerobjekte können auch selbst erstellt werden, was hin und wieder für selbst erstellte Knoten sinnvoll ist.

Die Wurzel der Replacerobjekte muss auch nicht die Wurzel des kompletten AST sein. Oft sogar wird nur ein Ausdruck oder eine Schleife verändert und im nachhinein wird diese in den restlichen AST eingefügt oder ersetzt etwas anderes.

Implementierung

Im Prinzip sind die Replacer ein zweiter Baum mit der gleichen Gestalt wie der AST in welchem Kindknoten ihre Elternknoten kennen. Dabei müssen nur zwei Pointer gespeichert werden, einer auf den eigentlichen AST Knoten und ein Pointer auf den Elternknoten.

Zusätzlich müssen Replacer noch ihre Position innerhalb den Elternknoten kennen. Also es muss sich für jeden Knoten gemerkt werden welches Kind am Elternknoten er ist, und wenn er Teil einer Knotenliste ist, welche Position er innerhalb dieser Liste hat[6]. Damit Knoten in Knotenlisten eingefügt und aus ihnen gelöscht werden können, wird in den Replacern, welche Knoten in Listen repräsentieren, spezieller Zustand mitgeführt um das Einfügen von mehreren Knoten relativ zu anderen zu ermöglichen.[7]

Beispiel

Abbildung 5.6 zeigt ein Beispiel dafür wie man den VisitorReplace einsetzen kann um alle Variablen in einem Programm zu finden und anschließend alle Variablen zufällig an eine andere Position tauscht.

[6] Sich eine Position in einer Liste zu merken ist einfach ein integer. Sich aber das jeweilige Kind zu merken geht in C++ nicht ohne weiteres. ALR generiert für jedes Kind an jedem Knotentyp eine neue Klasse (den Replacer für ein solches Kind) die weiß wie diese Kind ersetzt werden kann. In einer dynamisch typisierten Programmiersprache oder in einer Programmiersprache mit mehr Reflection könnte man das vermutlich ohne extra Typen lösen, wobei ich nicht weiss ob das besser ist.

[7] Um genauer zu sein werden einfach alle Einfüge- und Löschoperationen einer Liste gespeichert. Mit dieser Vorgeschichte und mit einem Zeitpunkt, an dem eine Position aktuell war, kann dann berechnet werden, wo diese Position zum aktuellen Zeitpunkt sein sollte.

idShuffler.cpp:

```
1  #include <random>
2  #include <algorithm>
3  #include "idShuffler.hpp"
4  #include "ast.hpp"
5
6  // Hilfsklasse, die den AST abläuft und dabei alle Identifier und alle
7  // Identifiern und deren Replacer speichert.
8  template<class TResult>
9  struct IdShufflerAnalyser : ast::VisitorReplace<TResult> {
10   template<class T>
11   using Replacer = std::shared_ptr<ast::ReplaceBase<TResult, T>>;
12
13   std::vector<ast::IdEx> ids;
14   std::vector<Replacer<ast::Ex>> replacer;
15   void visitReplaceIdEx(ast::IdEx id, Replacer<ast::Ex> replace)
16      final override {
17     ids.push_back(id);
18     replacer.push_back(replace);
19   }
20 };
21
22 ast::TUnitB shuffleIds(ast::TUnitB tUnit) {
23   auto root_replacer =
24      std::make_shared<ast::ReplaceStart<ast::TUnitB, ast::TUnitB>>(tUnit);
25
26   IdShufflerAnalyser<ast::TUnitB> analyser;
27   tUnit->visitReplace(analyser, root_replacer);
28
29   std::random_device rd; std::mt19937 g(rd());
30   std::shuffle(analyser.ids.begin(), analyser.ids.end(), g);
31
32   for (size_t i = 0; i < t.ids.size() & i < t.replacer.size(); ++i) {
33     analyser.replacer[i]->replaceThis(t.ids[i]);
34   }
35   return root_replacer->result;
36 }
```

Abbildung 5.6.: Die Funktion `shuffleIds` mischt all Vorkommnisse von Identifiern zufällig. In Zeile 33 werden die Replacer der Identifier genutzt, um diese mit der vermischten Variante zu ersetzen. Die neuen Wurzelknoten, die beim Ersetzen entstehen, werden im `root_replacer` gespeichert.

5.5. Codegenerierung

Das Resultat von ALR muss wieder OpenCL-C oder C Code sein und auch hier ist es praktisch den AST als Zwischenrepräsentation zu nutzen. Der AST stellt den Code bis auf Kleineres (wie Klammern und Whitespace) schon relativ klar dar.

Zur Erstellung des Ausgabecodes wird einfach nur für jeden Knoten im AST festgelegt , wie seine textuelle Repräsentation in Abhängigkeit der textuellen Repräsentationen der Kindknoten aussieht. Das Einzige, das nicht in dieses Schema, passt sind Typen, Deklarationen, Funktionen Definitionen. In C werden Typen von „Innen nach Außen“ gelesen und müssen so auch geschrieben werden. Zudem muss der Name eines deklarierten Identifiers mitten in den Typausdruck plaziert werden, obwohl dieser mit dem Typ erstmal nichts zu tun hat. In dem Fall von Deklarationen und Typen muss also der gesamte Teilbaum auf einmal betrachtet werden.

Im AST sind auch keine Informationen enthalten wo Klammern in Ausdrücken oder Typen gesetzt werden müssen oder sollen. Beim Generieren des Ausgabecodes für Ausdrücke werden einfach um jeden Teilausdruck Klammern gesetzt. Ohne Klammern kann es möglicherweise zu Verwechslungen führen, wenn dieser später in dem umschließenden Ausdruck eingebettet wird. Damit kann jeder Ausdruck separat betrachtet werden und „kleine“ Ausdrucke wie Literale oder Variablen können ohne Klammern ausgegeben werden. Da Teilbäume, die Typen repräsentieren, sowieso zusammen betrachtet werden, werden hier einige unnötige Klammern weggelassen.

Im AST fehlen auch jegliche Informationen wie der Code formatiert werden soll. Um die Darstellung etwas angenehmer zu machen, werden man manchen Stellen mechanisch Leerzeichen und Zeilenumbrüche eingefügt. Zudem wird die Einrückung von geschachtelten Codeblöcken beachtet. Ein Problem ist, dass die Zeilen oft zu lang sind und dabei Struktur nicht beachtet wird, die bei händisch geschriebenem Code, durch Aufteilung in mehrere Zeilen, zum Vorschein gekommen wäre.

Ein Beispiel für mit ALR erzeugten Code ist in Anhang A.

6. Beispiel Transformationen in ALR

In dieser Arbeit wurden drei unterschiedliche Transformationen mit ALR implementiert, die von leicht bis zu etwas aufwendiger reichen.

Bei allen folgenden Beispieltransformationen müssen alle Variablennamen innerhalb einer Funktion eindeutig sein. Konkret darf ein Variablenname nie unterschiedliche Bedeutungen in unterschiedlichen Teilen der Funktion haben.

Definition: Einfache Schleife Viele Transformationen beschäftigen sich mit der Iteration über einen Zahlenbereich. In C wird das durch eine Schleife dargestellt, diese könnten aber deutlich komplizierter sein als einfach nur eine Iteration über einen Zahlenbereich. Deshalb wird das Konzept einer einfachen Schleife eingeführt.

Eine einfache Schleife muss von der Form

```
1 for (int i = [[exp1]]; i < [[exp2]]; ++i) { ... }
```

sein, wobei anstatt `i` auch jeder andere Variablenname genutzt werden darf und die Schleifenobergrenze `[[exp2]]` und die Schleifenuntergrenze `[[exp1]]` ein Ausdruck sein muss. Die Schleifenvariable darf innerhalb der Schleife nicht geändert werden und der Ausdruck für `[[exp2]]` und `[[exp1]]` muss während des kompletten Schleifendurchlaufs konstant bleiben und darf keine Seiteneffekte haben.

6.1. Loop Unrolling

Loop Unrolling (Schleifen Ausrollen) ist eine Transformation die auch oft von Compilern angewendet wird. Trotzdem macht es Sinn von Hand dem Compiler ausgerollte Schleifen vorzusetzen, denn der Compiler entscheidet vielleicht fälschlicherweise, die Schleife nicht auszurollen. Es könnte auch sein, dass der Kompiler ungünstig häufig ausrollt und das, für die Hardware und das Problem, nicht optimal ist. Kontrolliert man also die Anzahl der Aufrollungen, kann man die Häufigkeit perfekt für ein Scenario einstellen.

Die Schleife, die ausgerollt werden soll, muss eine einfache Schleife sein und die Schleifenuntergrenze muss `0` sein. Ein `#pragma alr(1)unroll(#)` führt dazu, dass die Schleife so oft ausgerollt wird, wie in dem Parameter `#` angegeben wird.

Vorher:

```
1  #pragma alr(1) unroll(4)
2  for (int i = 0; i < size; ++i) {
3    result[i] = do_calculation(data1[i], data2[i]);
4  }
```

Nachher:

```
1  {
2    int i = 0;
3    for (; (i <= (size - 4)); (i += 4)) {
4      { (result[(i + 0)] = do_calculation(data1[(i + 0)], data2[(i + 0)])); }
5      { (result[(i + 1)] = do_calculation(data1[(i + 1)], data2[(i + 1)])); }
6      { (result[(i + 2)] = do_calculation(data1[(i + 2)], data2[(i + 2)])); }
7      { (result[(i + 3)] = do_calculation(data1[(i + 3)], data2[(i + 3)])); }
8    }
9    for (; (i < size); ++i) {
10     (result[i] = do_calculation(data1[i], data2[i]));
11   }
12 }
```

Abbildung 6.1.: Der Schleifenkörper wird mehrfach ausgeführt und die Schleifenvariablen werden entsprechend angepasst. Die dadurch nicht abgedekten Schleifendurchläufe werden am Ende einzeln gemacht.

Loop Unrolling führt einfach nur den Körper einer Schleife mehrfach aus bevor wieder zum Anfang der Schleife gesprungen wird, wie in Abbildung 6.1 beispielhaft gezeigt wird. Loop Unrolling kann mehrere Performanzvorteile haben. Zum Beispiel werden mehr Instruktionen pro Schleifenzyklus ausgeführt, was zum einen zu mehr Parallelisierung auf Instruktionebene führen kann und zum anderen wird der Vergleich für die Schleife seltener gemacht.

6.2. Vervielfältigung

Häufig müssen bei Optimierungen Statements vervielfacht werden. Diese Transformation soll dabei unterstützen, leicht Kopien von einem Statement zu erstellen. Durch ein `dup` Pragma (z.B. `#pragma alr(1)dup(d,4)`) wird spezifiziert, dass das auf das Pragma folgende Statement kopiert werden soll. Der zweite Parameter (hier `4`) ist eine ganze nicht negative Zahl und gibt an, wie viele Kopien von dem Statement erzeugt werden. Der erste Parameter (hier ein `d`) ist ein Identifier, der in den Kopien ersetzt wird. In der ersten Kopie wird er mit `0` ersetzt und in weiteren Kopien mit aufsteigenden Zahlen. Es werden nicht nur komplette Identifier ersetzt sondern auch Teile davon. Damit ein Teil von einem Identifier ersetzt wird, muss dieser durch Unterstriche vom Rest getrennt sein. Wird `d` in `data_d[d]` ersetzt, dann wird das erste `d` nicht ersetzt, das zweite und dritte `d` aber schon. In der ersten Kopie von `data_d[d]` wird daraus also `data_0[0]`. Die Identifier werden nur in Ausdrücken und dem

Vorher:

```
1 #pragma alr(1) dup(i,2) dup(d,2)
2 double cache_d_i = data_d[i];
```

Nachher:

```
1 double cache_00_00 = data_00[00];
2 double cache_00_01 = data_00[01];
3 double cache_01_00 = data_01[00];
4 double cache_01_01 = data_01[01];
```

Abbildung 6.2.: Deklaration und Initialisierung von acht Variablen mit der Vervielfältigungstransformation.

Namen der Deklaration ersetzt. Abbildung 6.2 zeigt beides. Wie man sieht, ist es möglich, mehrere Vervielfältigungstransformationen in einem Pragma zu spezifizieren. Bei den anderen Beispieltransformationen kann immer nur eine Transformation pro Pragma stehen.

Die Vervielfältigungstransformation auf einen Block oder eine Schleife anzuwenden hat die gleiche Wirkung, wie diese Transformation auf alle beinhalteten Statements anzuwenden. Für die beinhalteten Statements wird diese Regel auch wieder angewandt. Bei einer Schleife werden nur die Elemente innerhalb der Schleife kopiert, nicht aber der Schleifenkopf. Abbildung 6.3 stellt dazu ein Beispiel dar und zeigt, dass man Vervielfältigungstransformationen auch schachteln kann. Werden sie geschachtelt, dann verhalten sie sich so, wie wenn mehrere Vervielfältigungstransformationen angewendet werden.

6.3. Local Memory Caching

Die Workitems in OpenCL können, wenn die Hardware das unterstützt, in Gruppen zusammengefasst werden, sogenannte Workgroups [4]. Je nach Hardware steht einer Workgroup eine gewisse Menge an Speicher zur Verfügung, der als lokaler Speicher bezeichnet wird. Im Vergleich zum globalen Speicher, der allen Workitems zur Verfügung steht, ist der lokaler Speicher meist schneller[1] und hat allgemein andere Performanzcharakteristiken.

Es gibt also Fälle, in denen es sinnvoll ist, verwendete Daten im lokalem Speicher zwischenzuspeichern. Alle Workitems können gleichzeitig einen Wert aus dem globalen Speicher in den lokalen kopieren. Auf diese lokale Kopie können danach alle Workitems in der Workgroup zugreifen. Wollen alle Workitems über ein Array im globalen Speicher iterieren, dann kann die `localcache` Transformation dabei helfen, die Zwischenspeicherung im lokalen Speicher vorzunehmen.

[1] AMD macht diese Aussage zum Beispiel in [7] in Kapitel 2.10. Globaler Speicher hat unter den dort gezeigten Grafikkarten eine 11 bis 17 Mal geringere Höchstspeicherübertragung als lokaler Speicher (dort mit LDS bezeichnet).

Vorher:

```
1  kernel void main(global const double* data1, global const double* data2,
2     uint size, global double* result) {
3    int globalIdx = 2 * get_global_id(0);
4  #pragma alr(1) dup(k,2)
5    {
6      double accumulator_k = 0;
7      for (int i = 0; i < size; ++i) {
8        #pragma alr(1) dup(j,10)
9        accumulator_k += do_calculation(data1[j * size + i], data2[j * size + i],
             globalIdx + k);
10     }
11     result[globalIdx + k] = accumulator;
12   }
13 }
```

Nachher:

```
1  kernel void main(global const double* data1, global const double* data2, uint
       size, global double* result) {
2    int globalIdx = (2 * get_global_id(0));
3    {
4      double accumulator_00 = 0;
5      double accumulator_01 = 0;
6      for (int i = 0; (i < size); ++i) {
7        (accumulator_00 += do_calculation(data1[((00 * size) + i)], data2[((00 *
             size) + i)], (globalIdx + 00)));
8        (accumulator_01 += do_calculation(data1[((00 * size) + i)], data2[((00 *
             size) + i)], (globalIdx + 01)));
9        /* 16 weitere solche Zeilen */
10       (accumulator_00 += do_calculation(data1[((09 * size) + i)], data2[((09 *
             size) + i)], (globalIdx + 00)));
11       (accumulator_01 += do_calculation(data1[((09 * size) + i)], data2[((09 *
             size) + i)], (globalIdx + 01)));
12     }
13     (result[(globalIdx + 00)] = accumulator);
14     (result[(globalIdx + 01)] = accumulator);
15   }
16 }
```

Abbildung 6.3.: Code aus Abbildung 4.1 so abgewandelt, dass jedes OpenCL Workitem die Arbeit von zwei Workitems ausführt. Das passiert wegen des Pragmas in Zeile 4. Zudem wurde die innerste Schleife mit einer Vervielfältigungstransformation komplett ausgerollt, was möglich ist, wenn die Iterationsanzahl bekannt ist.

Die `localcache` Transformation kann nur auf einfache Schleifen angewendet werden, die im Folgenden als außenstehende Schleifen bezeichnet werden. Die außenstehende Schleife darf, bis auf einfache Schleifen mit jeweils unterschiedlichen Schleifenvariablen, keine weiteren Schleifen enthalten.

Die `localcache` Transformation hat drei Parameter. Der erste ist eine Zahl, die der Größe der Workgroup entspricht. Die Anzahl der Schleifeniterationen der außenstehenden Schleife muss durch diese Zahl teilbar sein. Der zweite Parameter ist eine Variable, welche den Index des Threads innerhalb der Gruppe enthält (aus der OpenCL Function `get_local_id` erhaltbar). Der dritte Parameter ist eine durch Leerzeichen getrennte Liste an Variablen, mit dem Typ **`double`**$*$. Wird innerhalb der Schleife nun eine Arrayindizierung in eine dieser Variablen gemacht, wird diese im lokalen Speicher für alle Workitems zwischengespeichert. Die Arrayindizierung muss auch für alle Workitems gleich sein.

Alle zwischengespeicherten Arrayzugriffe dürfen zudem nur Ausdrücke enthalten, die aus Variablen und Integerliteralen bestehen, welche durch die Operationen Plus (`+`), Minus (`-`), Mal($*$), Geteilt (`/`) und Rest (`%`) verknüpft werden. Mit der Ausnahme von Schleifenvariablen, dürfen sich diese Variablen während eines Durchlaufs der Schleife nicht ändern. In allen zwischengespeicherten Arrayindizierungen muss auch die Schleifenvariable der außenstehenden Schleife vorkommen und zwar so, dass in jeder Schleifeniterationen eine andere Stelle indiziert wird.

Überlappen sich die Arrayindizierungen einer Variablen, die zwischengespeichert wird, und sind die Arrayindizierungsausdrücke nicht identisch, dann werden die Werte der Arrayindizierungen separat und damit mehrfach zwischengespeichert.

Algorithmus Der Algorithmus, um den AST anzupassen, ist, im Vergleich zu den anderen beiden Beispieltransformationen, aufwendiger und weniger offensichtlich. Im Folgenden werde ich ihn erläutern. Ein Beispiel, in dem zumindest manche der Fälle auftreten, ist die Transformation aus dem Code in Abbildung 4.1 zu dem Resultat in Anhang A.

- Sammele aller Arrayzugriffe in zu zwischenspeichernden Variablen und alle geschachtelten Schleifen.
- Generiere eine neue innere Schleife die den gleichen Schleifenkörper hat wie die außenstehende Schleife.
- Ersetzte den Schleifenkörper der außenstehenden Schleife durch die neue innere Schleife.
- Alle Arrayzugriffe, die identisch sind, werden in Gruppen zusammengefasst. Identisch sind sie genau dann, wenn sowohl die indizierte Variable als auch der indizierende Ausdruck gleich sind.
- Für jede solche Gruppe wird folgendes gemacht:

Vorher:

```
1  int localIdx = get_local_id(0);
2  #pragma alr(1) localcache(64,localIdx,data1 data2)
3  for (int i = 0; i < size; ++i) {
4    result[i] = do_calculation(data1[i], data2[i]);
5  }
```

Nachher:

```
1  int localIdx = get_local_id(0);
2  for (int i = 0; (i < size); (i += 0)) {
3    local double cache_0_data1[64];
4    (cache_0_data1[localIdx] = data1[(i + localIdx)]);
5    local double cache_1_data2[64];
6    (cache_1_data2[localIdx] = data2[(i + localIdx)]);
7    barrier(CLK_LOCAL_MEM_FENCE);
8    for (int inner_i = 0; (inner_i < 64); (inner_i += 1)) {
9      (result[i] = do_calculation(cache_0_data1[inner_i], cache_1_data2[inner_i]));
10     ++i;
11   }
12   barrier(CLK_LOCAL_MEM_FENCE);
13 }
```

Abbildung 6.4.: Nach der Transformation wird in der äußeren Schleife von allen 64 Workitems genau ein Wert von `data1` und `data2` aus dem globalem Speicher in den lokalen Speicher zwischengespeichert.

- Finde alle involvierten Schleifen. Eine Schleife ist genau dann involviert (mit dieser Gruppe) wenn ihre Schleifenvariable in dem Indizierungsausdruck vorkommt. Betrachtet werden hier nur die Schleifen die geschachtelt in der zu optierenden ausstehenden Schleife sind. Das Bedeutet, dass auch zum Zwischenspeichern diese Schleifen durchlaufen werden müssen.
- Füge folgende generierte Statements innerhalb der außenstehenden Schleife und vor der neuen inneren Schleife ein:
  - Generieren des Ausdrucks der Angibt wie viele Elemente Zwischengespeichert werden müssen. Das ist die Größe der Workgroup multipliziert mit dem Produkt aus den Iterationanzahlen aller involvierten Schleifen.
  - Generiere die Deklaration der Zwischenspeichervariable. Hierbei handelt es sich um das Array das den Zwischenspeicher darstellt. Die Größe des Arrays ist durch den Ausdruck aus dem letzten Schritt gegeben.
  - Alle involvierten Schleifen geschachtelt generieren und darin den Zwischenspeicher zuweisen. Hier weist jedes Workitem nur einen Wert pro Durchlauf der innersten Schleife zu. Die Linke Seite der Zuweisung ist die Zwischenspeichervariable die mit den Schleifenvariablen und Iterationsanzahlen der involvierten

Schleifen sowie dem Index der Workitem indiziert ist. Diese Rechte Seite wird leicht Abgewandelt im nächsten Schritt wieder beim lesen des caches genutzt. Die Rechte Seite der Zuweisung ist genau der Arrayindizierungsausdruck wie er vom Programmierer genutzt wurde nur wird der Schleifenindex der außenstehenden Schleife durch den Index des Workitem ersetzt. Der Schleifenindex der außenstehenden Schleife ist der zweite Parameter für diese Transformation.

– In ursprünglichen Schleifenkörper alle Arrayindizierungen abwandeln, so dass der Zwischenspeicher korrekt genutzt wird. Hier benutzen wir den Ausdruck aus dem letzten Schritt und ersetzten den Index der Workitems durch die Schleifenvariable der neuen inneren Schleife.

6.4. Laufzeit Tests

Um qualitativ zu zeigen, dass die Beispieltransformationen eine Laufzeitverbesserung erbringen können, wurden sie auf Beispielcode angewandt, der mit und ohne diese Transformationen zur Verfügung stand.

Bei dem Beispiel handelt es sich um Data-Mining auf dünnen Gittern. Genauer gesagt wird eine Regressionaufgabe auf den Funktionen gelöst, welche auf einem regulären zehndimensionalen dünnen Gitter mit 6 Ebenen dargestellt werden können. Der Datensatz, für den das Regressionsproblem gelöst wird, ist ein Friedmann 10d Datensatz mit 150000 Datenpunkten. Näheres zu Regression auf dünnen Gittern und dem Friedmann 10d Datensatz kann in [8] gefunden werden.

Es gibt eine Referenzimplementierung, die Code für dieses Beispiel mit unterschiedlichen Optimierungen generieren kann. Diese Referenzimplementierung kann auch die Schleife ausrollen und die Local Memory Cache Transformation anwenden. Der Beispielcode hat eine ähnliche Struktur wie der Code in Abbildung 4.1: Schleifen und darin enthaltene Speicherzugriffe auf den globalen Speicher sind gleich. In der Referenzimplementierung werden die Local Memory Caching und Ausroll Transformation an denselben Schleifen angewandt wie in Abbildung 4.1.

Die Referenzimplementierung liefert ohne angewandte Transformation den Code, welcher von ALR Transformiert wird. Dieser Code ist nur minimal geändert. Zum einen werden die Pragmas hinzugefügt, die ALR sagen, wo welche Transformationen angewendet werden sollen. Zum anderen wird eine Hilfsvariable entfernt, indem sie durch ihre Definition ersetzt wird. Diese Hilfsvariable würde die Anwendung der Local Cache Transformation von ALR verhindern.

Die folgenden vier Szenarien wurden mit der Referenzimplementierung und ALR getestet.

1. Ohne Transformationen
2. Mit Local Memory Cache Transformation

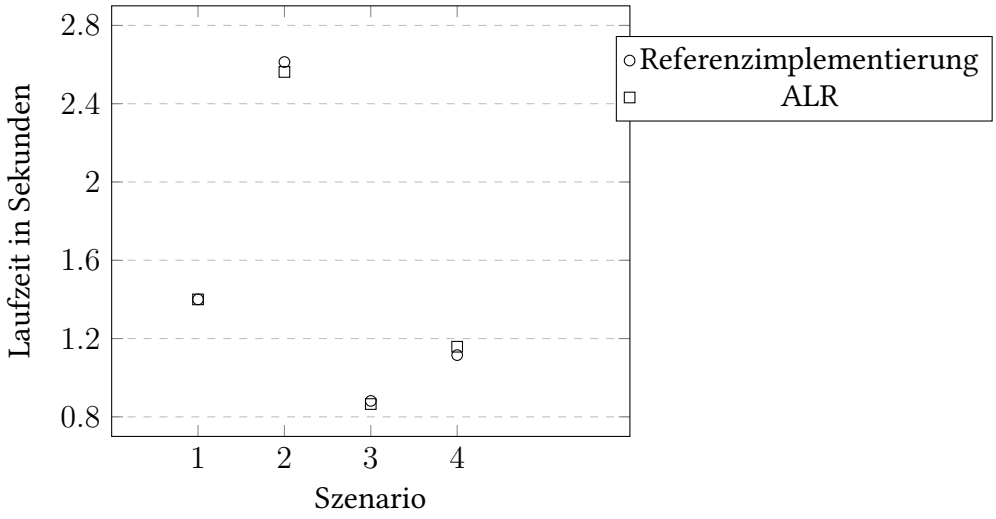


Abbildung 6.5.: Der Beispielcode ausgeführt auf einer AMD FirePro W8100 (FireGL V).

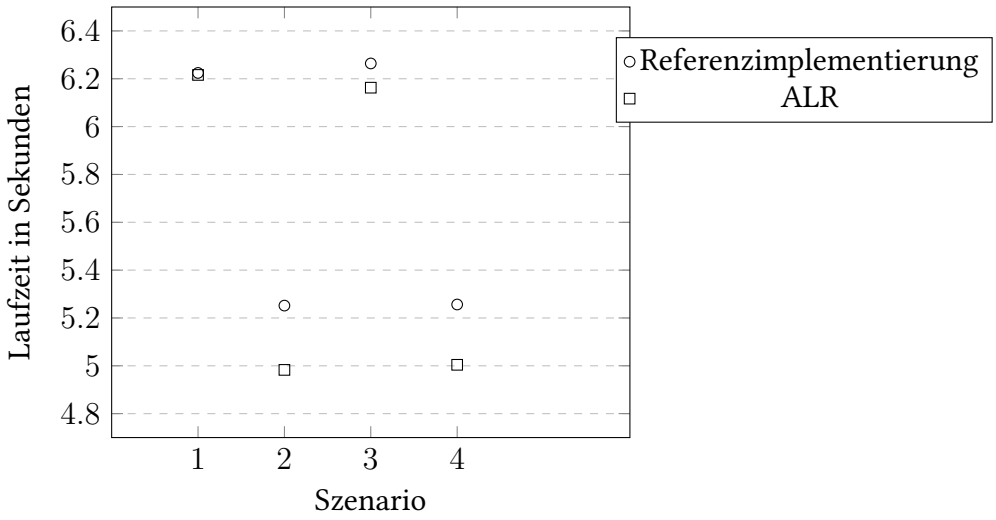


Abbildung 6.6.: Der Beispielcode ausgeführt auf einer NVIDIA GeForce GTX 680.

3. Mit Ausroll Transformation
4. Mit Local Memory Cache und Ausroll Transformation

Abbildungen 6.6 und 6.5 zeigen, dass sich die Transformationen sehr unterschiedlich auf den beiden Hardwareplattformen verhalten. Außerdem ist ersichtlich, dass die Transformationen von ALR mit Optimierungen mithalten können, welche speziell auf den Beispielcode zugeschnitten sind, wie das in der Referenzimplementierung der Fall ist.

7. Ausblick und Zusammenfassung

Ein anstrebenswürdiges Ziel währe vielleicht eine Metasprache, welche die unterliegende Programmiersprache ergänzt und primitive bis aufwendige Transformationen bereitstellt, wobei die Transformationen unter sich gut kombinierbar sind.

Mit einer aufwendiger Transformation ist gemeint, dass die Transformation viele Teile involviert. Local Memory Caching aus Kapitel 6.3 macht zum Beispiel viele Unterschiedliche Actionen: Neue Variablen deklarieren und auf komplizierte Weise initialisieren, an unterschiedlichen Stellen Ausdrücke austauschen und Schleifen Manipulieren. Ein grosser Bereich wird in vielen hinsiechten manipuliert. Solche aufwendigen Transformationen haben mit wenig Aufwand für den Programmierer das Potenzial für akzeptable Performanzverbesserungen. Besonders gute Laufzeitverbesserungen mit solchen Transformationen zu erzieht erfordert ist schwer oder erfordert vielleicht sehr hohen Implementierungsaufwand für den Implementer der Transformation um die nötige Flexibilität Bereitzustellen.

Die Local Memory Caching Beispieltransformation lässt einiges an Flexibilität zu wünschen übrig. Manche davon wären wohl trivial implementierbar, wie die Unterstützung weitere Typen (neben **`double`**s) zwischenzuspeichern, andere Flexibilitäten ware aber deutlich komplizierter zu implementieren. Zum Beispiel wäre es viel Aufwendiger dem Programmierer mehr Möglichkeiten zu geben wie die Speicheranordnung (memory layout) der Caches aussehen soll. In Beiden gerade genannten Beispielen würde auch die zusätzlichen und komplizierteren Parameter an dieser Optimierung zu einer Verkomplizierung der Metasprache führen.

Die Vervielfältigungs Transformation aus Kapitel 6.2 war der Versuche eine „mächtige“ primitive Transformation zu bauen. Für die meisten Anwendungen davon wären aber noch weitere primitive Transformation von Nöten. Vor allem konditionale primitive Transformationen und vielleicht Variablen ähnlich zu denen wie sie im Präprozessor vorkommen. Mit gut zusammenpassenden primitiven Transformationen, die viele Möglichkeiten zur Verfügung stellen lassen sich vermutlich viele Unterschiedliche Programme generieren, die gut für Autotuning verwendet werden können. Das schwierige ist gut zusammenpassenden primitiven Transformationen zu definieren und entwickeln. Außerdem ist es wohl ein großes Problem diese Programme übersichtlich zu halten. Vermutlich hilft es sich von Pragmas zu entfernen und wie der Präprozessor die Metasprache direkt nach **`#`** zu setzten (zum Beispiel **`#if`**).

Sowohl aufwendigere als auch primitivere Transformationen könnten eine große Unterstützung bei der Entwicklung von performanter Software darstellen. Die Ergebnisse dieser Arbeit zeigen zumindest, dass dies mit wenigen Transformationen in keinen Beispielen gelingen

kann. Wie die Komplexität davon aber in grösseren Szenarien, bei denen mehrere solcher Transformationen interagieren ist ungewiss.

Zusammenfassend wurde in dieser Arbeit ein Source-to-Source Compiler entwickelt, für den einige wenige Transformationen implementiert wurden, welche bei der Laufzeitoptimierung von Software unterstützen können.

A. Transformationsresultat des Beispielcodes

Das Resultat des Codes aus Abbildung 4.1, nachdem er mit den Beispieltransformationen aus dem Kapitel 6 transformiert wurde.

```
1  kernel void main(global const double* data1, global const double* data2, uint
       size, global double* result) {
2    int globalIdx = get_global_id(0);
3    int localIdx = get_local_id(0);
4    double accumulator = 0;
5    for (int i = 0; (i < size); (i += 0)) {
6      local double cache_0_data1[(64 * (10 - 0))];
7      for (int j = 0; (j < 10); ++j) {
8        (cache_0_data1[(localIdx + (64 * j))] = data1[((j * size) + (i + localIdx))]);
9      }
10     local double cache_1_data2[(64 * (10 - 0))];
11     for (int j = 0; (j < 10); ++j) {
12       (cache_1_data2[(localIdx + (64 * j))] = data2[((j * size) + (i + localIdx))]);
13     }
14     barrier(CLK_LOCAL_MEM_FENCE);
15     for (int inner_i = 0; (inner_i < 64); (inner_i += 1)) {
16
17 #pragma alr(2) unroll(10)
18       {
19         int j = 0;
20         for (; (j <= (10 - 10)); (j += 10)) {
21           {
22             (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * (j +
                   0)))], cache_1_data2[(inner_i + (64 * (j + 0)))], globalIdx));
23           }
24           {
25             (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * (j +
                   1)))], cache_1_data2[(inner_i + (64 * (j + 1)))], globalIdx));
26           }
27           {
28             (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * (j +
                   2)))], cache_1_data2[(inner_i + (64 * (j + 2)))], globalIdx));
29           }
```

```
30          {
31            (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * (j +
                  3)))], cache_1_data2[(inner_i + (64 * (j + 3)))], globalIdx));
32          }
33          {
34            (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * (j +
                  4)))], cache_1_data2[(inner_i + (64 * (j + 4)))], globalIdx));
35          }
36          {
37            (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * (j +
                  5)))], cache_1_data2[(inner_i + (64 * (j + 5)))], globalIdx));
38          }
39          {
40            (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * (j +
                  6)))], cache_1_data2[(inner_i + (64 * (j + 6)))], globalIdx));
41          }
42          {
43            (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * (j +
                  7)))], cache_1_data2[(inner_i + (64 * (j + 7)))], globalIdx));
44          }
45          {
46            (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * (j +
                  8)))], cache_1_data2[(inner_i + (64 * (j + 8)))], globalIdx));
47          }
48          {
49            (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * (j +
                  9)))], cache_1_data2[(inner_i + (64 * (j + 9)))], globalIdx));
50          }
51        }
52        for (; (j < 10); ++j) {
53          (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * j))],
                cache_1_data2[(inner_i + (64 * j))], globalIdx));
54        }
55      }
56      ++i;
57    }
58    barrier(CLK_LOCAL_MEM_FENCE);
59  }
60  (result[globalIdx] = accumulator);
61 }
```

Dasselbe wie oben aber ohne die Loop Unrolling Transformation:

```
1 kernel void main(global const double* data1, global const double* data2, uint
      size, global double* result) {
2   int globalIdx = get_global_id(0);
3   int localIdx = get_local_id(0);
```

```
4   double accumulator = 0;
5   for (int i = 0; (i < size); (i += 0)) {
6     local double cache_0_data1[(64 * (10 - 0))];
7     for (int j = 0; (j < 10); ++j) {
8       (cache_0_data1[(localIdx + (64 * j))] = data1[((j * size) + (i + localIdx))]);
9     }
10    local double cache_1_data2[(64 * (10 - 0))];
11    for (int j = 0; (j < 10); ++j) {
12      (cache_1_data2[(localIdx + (64 * j))] = data2[((j * size) + (i + localIdx))]);
13    }
14    barrier(CLK_LOCAL_MEM_FENCE);
15    for (int inner_i = 0; (inner_i < 64); (inner_i += 1)) {
16
17 #pragma alr(2) unroll(10)
18      for (int j = 0; (j < 10); ++j) {
19        (accumulator += do_calculation(cache_0_data1[(inner_i + (64 * j))],
              cache_1_data2[(inner_i + (64 * j))], globalIdx));
20      }
21      ++i;
22    }
23    barrier(CLK_LOCAL_MEM_FENCE);
24  }
25  (result[globalIdx] = accumulator);
26 }
```

B. AST Knotentypen in ALR

```
StrLit:Ex        | v^str                            // string literal
IntLit:Ex        | v^str                            // int literal
CharLit:Ex       | v^str                            // char literal
FloatLit:Ex      | v^str                            // float literal

IdEx:Ex          | id^str                           // identifier
ArrEx:Ex         | ex:Ex idx:Ex                     // array indexing
CallEx:Ex        | ex:Ex args@Ex                    // function call
DotEx:Ex         | ex:Ex id^str                     // member access
PtrEx:Ex         | ex:Ex id^str                     // pointer member access
BinEx:Ex         | op^str left:Ex right:Ex          // binary operation
PostEx:Ex        | ex:Ex op^str                     // postfix operation
PreEx:Ex         | ex:Ex op^str                     // prefix operation
CastEx:Ex        | ex:Ex ty:Ty                      // cast operation
TernaryEx:Ex     | cond:Ex iftrue:Ex iffalse:Ex     // ternary operation
TyEx:Ex          | ty:Ty                           // type expression
InitEx:Ex        | exs@Ex                          // initializer list

ForSt:St         | init@DeclB cond:Ex inc?:Ex st:St // for statement
WhileSt:St       | cond:Ex st:St                    // while statement
DoWhileSt:St     | cond:Ex st:St                    // do-while statement
IfSt:St          | cond:Ex st:St else_?:St          // if statement
SwitchSt:St      | ex:Ex st:St                      // switch statement
LblSt:St         | lbl^str st:St                   // labled statement
CaseSt:St        | cond:Ex st:St                    // case statement
DefaultSt:St     | st:St                           // default statement
EmptySt:St                                        // empty statement
BreakSt:St                                        // break statement
ContinueSt:St                                     // continue statement
ReturnSt:St      | ex?:Ex                          // return statement
GotoSt:St        | lbl^str                         // goto statement
BlkSt:St         | sts@St                          // block statement
ExSt:St          | ex:Ex                           // expression statement
DeclSt:St        | decls@DeclB                     // declaration statement
PragmaSt:St      | txt^str                         // pragam statement

QualTy:Ty        | quals^strVec ty:Ty               // qualified type
```

```
IdTy:Ty          | id^str                          // identifier type
StructTy:Ty      | tag^str decls@DeclB             // struct type
FuncTy:Ty        | retTy:Ty params@DeclB           // function type
PtrTy:Ty         | ty:Ty                          // pointer type
ArrTy:Ty         | ty:Ty arrLength?:Ex             // array type

Decl:DeclB       | ty:Ty name^str init?:Ex         // declaration

Func:TItem       | ty:Ty name^str body:St          // function definition
GlobalDecl:TItem | ty:Ty name^str init?:Ex         // global variable

TUnit:TUnitB     | items@TItem                     // translation unit
```

Wie diese zu verstehen sind, ist in Abbildung 5.2 erläutert.

Literaturverzeichnis

[1] *C Programming Language - Version 11* (zitiert auf S. 19).

[2] *C11 Definitionen für Flex*. URL: http://www.quut.com/c/ANSI-C-grammar-l-2011.html (zitiert auf S. 25).

[3] *C11 Grammatik für Bison*. URL: http://www.quut.com/c/ANSI-C-grammar-y-2011.html (zitiert auf S. 25).

[4] K. O. W. Group. *The OpenCL Specification - Version 1.2*. 2012. URL: https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf (zitiert auf S. 9, 19, 24, 25, 35).

[5] M. Güting Ralf Hartmut und Erwig. *Übersetzerbau - Techniken, Werkzeuge, Anwendungen*. Springer-Verlag, Berlin Heidelberg, 1999 (zitiert auf S. 13).

[6] D. M. Kernighan Brian W. und Ritchie. *The C Programming Language*. Englewood Cliffs, NJ: Prentice Hall, 1978 (zitiert auf S. 23).

[7] NVIDIA. *AMD APP SDK - OpenCL Optimization Guide*. 2015. URL: developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/ (zitiert auf S. 35).

[8] D. Pfander, A. Heinecke, D. Pflüger. „Sparse Grids and Applications - Stuttgart 2014“. In: Hrsg. von J. Garcke, D. Pflüger. Cham: Springer International Publishing, 2016. Kap. A New Subspace-Based Algorithm for Efficient Spatially Adaptive Sparse Grid Regression, Classification and Multi-evaluation, S. 221–246. URL: http://dx.doi.org/10.1007/978-3-319-28262-6_9 (zitiert auf S. 39).

Alle URLs wurden zuletzt am 30. 05. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift