gpu4java

Studienarbeit

Studiengang Informatik
OST – Ostschweizer Fachhochschule
Campus Rapperswil-Jona

Herbstsemester 2021

Autor(en):	Marc Emch, Floris Staub
Betreuer:	Prof. Dr. Mitra Purandare
Experte:	Philipp Kramer

Abstract

Prozessoren sind darauf ausgerichtet, viele komplexe Operationen sequenziell auszuführen. Auch moderne Multicore-Prozessoren ermöglichen nur eine geringe Parallelisierung. Dem gegenüber steht die Grafikkarte. Diese kann lediglich simple arithmetisch Operationen ausführen, hat jedoch tausende von Threads, die parallel ausgeführt werden können. Dadurch ist es möglich, sehr rechen-intensive Programme, wie zum Beispiel Finanzmarkt-Simulationen, stark zu beschleunigen.

In vielen Programmiersprachen ist es möglich, mit geringem Aufwand des Entwicklers Teile des Rechenaufwandes auf die Grafikkarte auszulagern. Java bietet jedoch von sich aus keine solche Möglichkeit. Da Java immer noch eine der meistverbreiteten Programmiersprachen ist, hat diese Arbeit das Ziel, eine Lösung zu entwickeln, die es einem Programmierer ermöglicht, Teile eines Java-Programms auf der Grafikkarte auszuführen. Dabei soll möglichst wenig zusätzlicher Aufwand für den Programmierer entstehen.

In dieser Arbeit wurde ein Tool entwickelt, welches markierte Teile eines Java-Programms nimmt und in CUDA-Code übersetzt, sodass diese auf Nvidia-Grafikkarten ausgeführt werden können. Dabei wird Java Bytecode als Input genommen, die markierten Teile werden ausgeschnitten und durch einen “native” Funktionsaufruf ersetzt. Diese native Funktion wird dann als C Source Code erzeugt, und zusammen mit dem CUDA-Code in eine Shared Library kompiliert, welche letztendlich von der Java Virtual Machine geladen und an die Klasse angebunden wird, aus welcher der Code ursprünglich ausgeschnitten wurde.

Danksagung

An dieser Stelle wollen wir uns bei den Personen bedanken, die uns durch die Arbeit geholfen haben. Wir danken besonders unseren Unterstützern die diese Arbeit möglich gemacht haben.

Mitra Purandare danken wir für die Betreuung der Studienarbeit, für ihr Engagement, und für die vielen hilfreichen Tipps.

Phillip Kramer danken wir für das Bereitstellen der Infrastruktur und seine Inputs während der Realisation.

Gerard Basler danken wir für die hilfreichen Inputs und das Bereitstellen von realen Anwendungs-Beispielen.

Inhaltsverzeichnis

1. Management Summary

1.1 Ausgangslage

Code auf einer Grafikkarte ausführen zu können, ist mit Java ein grösserer Aufwand als mit anderen Programmiersprachen. Es gibt bereits einige Lösungen wie Aparapi und TornadoVM, die es ermöglichen, Java Code auf einer Grafikkarte ausführen zu können. Diese Lösungen erfordern jedoch einen erheblichen zusätzlichen Aufwand für den Programmierer. In anderen Programmiersprachen wie C, C++ oder C# ist die Nutzung der Grafikkarte sehr viel einfacher. Für C# gibt es das Altimes Projekt, das es ermöglicht nativen C# Code mit Annotationen zu versehen um diese Teile parallelisiert auf der Grafikkarte ausführen zu können. Ein solches Projekt gibt es nicht für Java und das Ziel dieser Arbeit ist es, einen Grundbaustein dafür zu legen.

1.2 Vorgehen

Die Aufgabe war sehr offen und wir hatten grossen Freiraum mit der Auslegung des Projektes. Als ersten Schritt in dieser Arbeit mussten wir die Möglichkeiten austesten. Es gibt zwei grosse APIs für C/C++ mit welchen man Code für die Grafikkarte schreiben kann. Diese beiden APIs sind OpenCL, welches die meisten gängigen Grafikkarten unterstützt, und CUDA für Nvidia Grafikkarten.

Um uns mit der Materie auseinanderzusetzen, schrieben wir ein Proof of Concept Programm. Wir schrieben ein Java Programm, das OpenCL verwendet, um eine Matrixmultiplikation durchzuführen. Wir wollten unser erstes Proof of Concept in OpenCL schrieben, da man somit nicht von einem Hersteller abhängig ist. Im Laufe der Arbeit haben wir uns jedoch dazu entschieden, auf CUDA zu wechseln. CUDA ist sehr gut dokumentiert und weit verbreitet. Dazu wurde uns von der OST Hardware zur Verfügung gestellt. Wir hatten eine Nvidia GTX 1050 Ti, um unser Programm zu testen.

Nach dem Proof of Concept konnten wir wählen, ob wir eine Monte-Carlo-Simulation mit CUDA optimieren wollen, oder ein Bytecode Modifikation Tool schreiben wollen. Wir haben uns für Letzteres entschieden, und haben die Meilensteine definiert, welche im Kapitel "Zeitplan / Meilensteine" einsehbar sind. Wir haben damit begonnen, einen Bytecode Parser zu schreiben, welchen die Markierten Stellen im Code findet. Danach haben wir einen Data Flow Graph und Control Flow Graph implementiert. Zum Schluss haben wir noch den Java zu CUDA Transpiler implementiert.

1.3 Ergebnisse

Das Ergebnis dieser Arbeit ist ein Tool, welches es ermöglicht, markierte Teile in einem Java Code auf der Grafikkarte ausführen zu können. Als Ziel haben wir uns gesetzt, for-Loops und Arrays zu unterstützen, was wir auch erreicht haben. Es können beliebig komplexe Instruktionen unterstützt werden, jedoch werden zurzeit lediglich primitive Datentypen und eindimensionale Arrays unterstützt. Zum Ende dieser Arbeit haben wir keinen Speedup durch das Verwenden einer Grafikkarte erreicht. Dies liegt primär daran, dass das Kopieren von Daten auf die Grafikkarte und wieder zurück nicht optimiert wurde.

1.4 Ausblick

Wir haben in dieser Arbeit eine grundlegende Struktur geliefert, um das Problem aus der Aufgabenstellung zu lösen. Aus Zeit- und Komplexitätsgründen hat unsere Arbeit eine Reihe an Limitationen, welche unter "2.7 Limitationen" eingesehen werden kann. Jedoch stellen keine von diesen ein grundlegendes Problem dar, und könnten alle mit entsprechendem Aufwand behoben werden.

2. Technischer Bericht

2.1 Problemstellung

Es gibt diverse Anwendungen, die durch Parallelisierung mittels einer Grafikkarte stark beschleunigt werden könnten. Um dies in Java umsetzen zu können, muss jedoch ein grosser Aufwand betrieben werden. Man muss Code für die CPU und die GPU schreiben, und dies sollte optimalerweise in C oder C++ gemacht werden, da diese eine API zur Nutzung der Grafikkarten haben. Java ist eine der meistverbreiteten Programmiersprachen und wird für zahlreiche Enterprise Anwendungen verwendet. Für diese ist der Umstieg auf manuell geschriebenen GPU-Code nicht vertretbar. In dieser Arbeit soll daher ein Tool entwickelt werden, welches die Nutzung einer Grafikkarte mit Java vereinfacht.

2.2 Vision

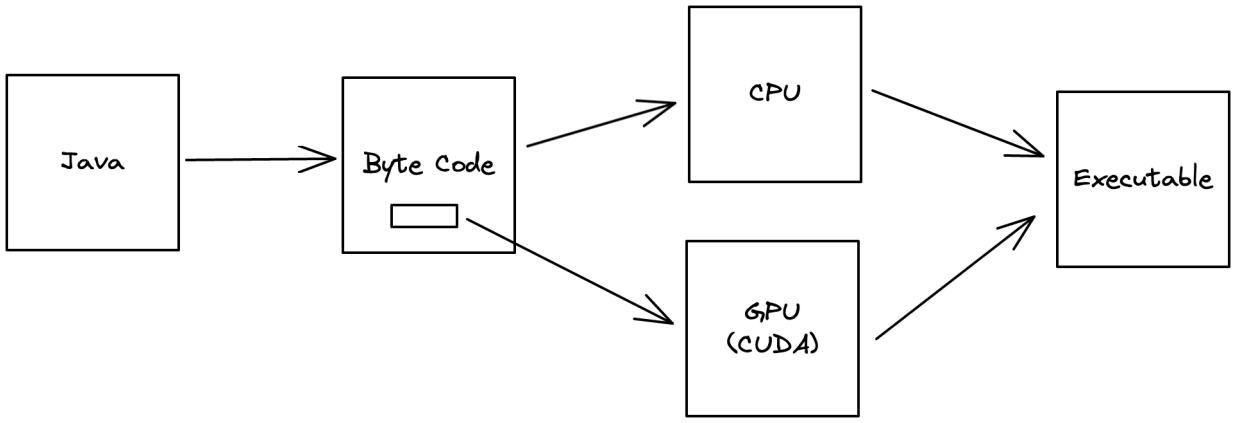

Abbildung 1 Vision Architektur

Die Vision dieser Arbeit ist es, dass Teile eines Java Codes mit wenig Extraaufwand von Seiten des Programmierers auf einer Nvidia Grafikkarte ausgeführt werden können. Es soll ein Tool entwickelt werden, welches diese Teile durch Annotationen findet und automatisch in Code für eine Grafikkarte übersetzt. Altimesh[1] ist ein Projekt, welches diese Problemstellung für die Programmiersprache C# realisiert. Dieses Projekt kann als Inspiration dienen für ein ultimatives Endziel. Es ist nicht realistisch, dass ein solches Tool in einer Studien- oder Bachelorarbeit realisiert werden kann. Jedoch kann ein Grundbaustein gelegt werden, auf welchem zukünftige Arbeiten aufbauen können. Als Ziel für diese Arbeit haben wir uns gesetzt, dass das Tool einfache for-loops mit Arrays unterstützen können soll.

[1] https://www.altimesh.com

2.3 GPU vs CPU

CPUs sind allgemeine Prozessoren, welche dafür ausgelegt sind, komplexe Control Fows, memory management und vieles mehr zu bearbeiten. Sie sind jedoch nicht für Parallelisierung ausgelegt. Selbst die modernsten Prozessoren haben eine sehr geringe Anzahl Cores. CPUs haben normalerweis 4, 8, 16 oder 64 Cores wobei jeder Core sehr unabhängig agieren kann und einen eigenen Cache hat.

GPUs haben eine viel höheren Gesamtdurchsatz und Memory Bandbreite als eine CPU für einen vergleichbaren preis und Stromverbrauch. Dies liegt daran, dass GPUs dafür entworfen sind, tausende von Threads parallel auszuführen, um für die schwächere Single-thread Leistung zu kompensieren. Dies wird erreicht, indem mehr Transistoren für Recheneinheiten verwendet werden als bei CPUs [1]. GPUs funktionieren nach dem Prinzip Single Instruction Multiple Data (SIMD). Das heisst, dass die Cores die gleichen Instruktionen auf unterschiedliche Daten anwenden. Cores werden in Gruppen aufgeteilt mit einem Control und L1 Cache für die ganze Gruppe.

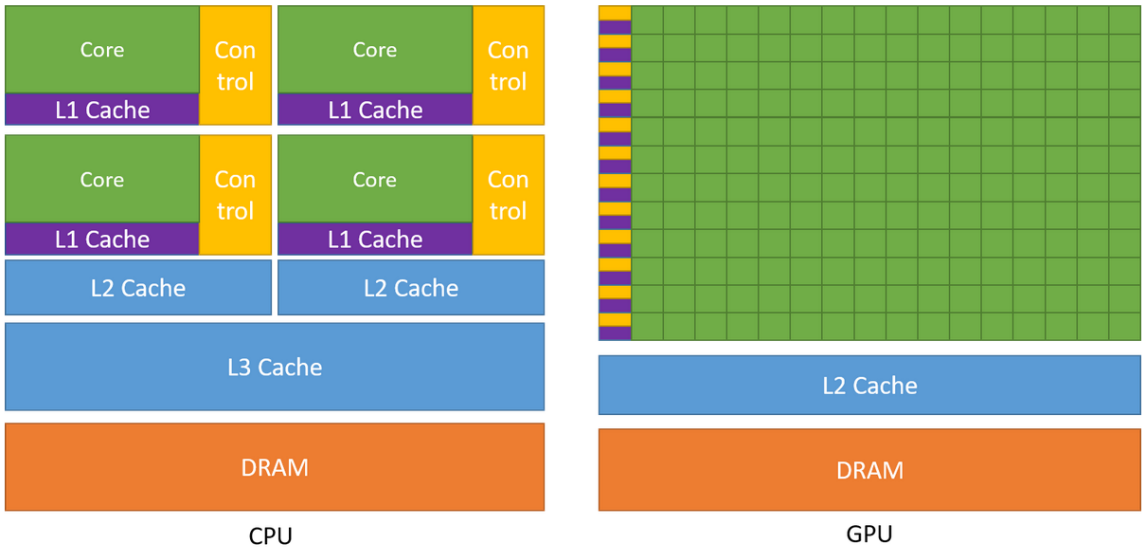

Abbildung 2 CPU vs GPU [2]

2.4 Eingesetzte Technologien

2.4.1 Java

Fast unsere ganze Arbeit ist in Java geschrieben. Prinzipiell hätten wir eine beliebige Programmiersprache für den Transpiler wählen können, aber da sich unsere Arbeit auf Java bezieht, und da die meisten Frameworks die mit Java Bytecode arbeiten in Java selbst geschrieben sind, war Java für uns die naheliegendste Wahl.

Unsere API und die Tests mussten so oder so in Java geschrieben werden.

2.4.2 ObjectWeb ASM

ObjectWeb ASM ist ein weit verbreitetes Java-Framework, welches das Analysieren und Modifizieren von Java-Klassen ermöglicht, ohne dass der Programmierer selbst Java Bytecode parsen muss. Dies vereinfachte unsere Arbeit nicht nur, sondern erlaubt es uns auch, eine grosse Anzahl an Java-Versionen zu unterstützen, da die Versionsunterschiede von ASM weg-abstrahiert werden.

2.4.3 JNI

Um eine Brücke zwischen Java und anderen Sprachen herzustellen (in unserem Fall GPU-Sprachen), bietet Java das "Java Native Interface" (JNI). Dabei deklariert man in Java eine Methode mit dem "native" Keyword, und kann dann in C oder einer anderen Sprache eine Implementation für diese Methode schreiben. Wir haben C benutzt, da dies die vom JNI "vorgesehene" Sprache ist und weil alle uns bekannten GPU-Libraries eine Anbindung an C oder C++ besitzen, wodurch wir einfache Interoperabilität erlangen. In unserem Fall wurde der C Code jedoch nicht von uns manuell geschrieben, sondern wird zu Laufzeit von unserem Transpiler generiert und direkt an den C Compiler des Systems weitergegeben.

2.4.4 CUDA C

Als Ziel-Hardware für den GPU-Code haben wir uns für Nvidia-Grafikkarten entschieden, wodurch die Wahl der Programmiersprache auf CUDA C fiel. Dieser CUDA C-Code wird zusammen mit dem C-Code des JNIs von unserem Transpiler generiert und an den C Compiler weitergegeben.

Wir haben uns für Nvidia-Hardware entschieden, da einerseits CUDA eine verhältnismässig einfache und gut dokumentierte Schnittstelle bereitstellt, und andererseits die OST bestehende Nvidia-Hardware hatte, die wir nutzen konnten.

2.4.5 Diskutierte Alternativen

Eine alternative zum Transpiler-Ansatz, die in der Planungsphase zur Debatte stand, wäre gewesen, eine bestehende Open-source Java Virtual Machine so zu modifizieren, dass zu Laufzeit des Java-Programms markierte Teile davon dynamisch in GPU-Code übersetzt und auf die GPU ausgelagert würden. In diesem Fall hätten wir unsere Arbeit in der selben Programmiersprache geschrieben, in der die gewählte JVM geschrieben ist. Dadurch wären weder ObjectWeb ASM noch JNI zum Einsatz gekommen, und der grösste Teil unserer Arbeit wäre sehr wahrscheinlich in C geschrieben worden.

Wir haben uns jedoch gegen diesen Ansatz entschieden, da wir uns in eine enorme bestehende Code-Basis hätten einarbeiten müssen, der Nutzer auf eine einzige JVM eingeschränkt worden wäre, und es sehr viel Aufwand bedeutet hätte, diese Lösung "up to date" mit der JVM zu halten. Im Kontrast dazu ist unser Transpiler ein in sich geschlossenes Programm, welches sowohl zu Compile- wie auch zur Laufzeit unabhänging von der benutzten JVM ist.

Ebenfalls stand zur Debatte, ob wir als GPU-Technologie OpenCL nutzen wollten, was uns eine grössere Freiheit in der Wahl der GPU-Hardware gelassen hätte. Nachdem wir jedoch unser Proof of Concept mit

OpenCL implementiert hatten, kamen wir zum Schluss, dass CUDA die robustere und besser zu handhabende Wahl war.

2.5 Proof of Concept

Unser erstes Proof of Concept bestand aus einer einfachen 3x3 Matrix-Multiplikation. Dabei haben wir in Java ein float-Array mit Zahlen gefüllt, dieses an eine "native" Methode übergeben, und danach alle Elemente an die Konsole ausgegeben. Die in C geschriebene native Methode hat das Java-Array in ein C-Array (bzw. Pointer) umgewandelt, dieses an die GPU angebunden, und einen vorprogrammierten OpenCL-Kernel auf der GPU ausgeführt.

2.6 Implementation

2.6.1 Transpiler

Die grundlegende Funktionsweise unserer Arbeit ist als Transpiler: Eine Java JAR-Datei wird an den Transpiler übergeben, und der Transpiler liefert eine modifizierte JAR-Datei und eine Shared Library zurück. Daneben steht eine minimale API-Klasse zur Verfügung, um Code-Abschnitte zu markieren, die auf die Grafikkarte ausgelagert werden sollen. Diese Klasse stellt zwei statische Methoden zur Verfügung, `enterGPU` und `exitGPU`. Diese dienen nur als Markierungen und machen in der Java-Implementation absolut nichts. Dadurch hat man im Zweifelsfall immer die Option, den Transpiler nicht zu benutzen, und stattdessen die unmodifizierte JAR-Datei auszuführen. Dies ist die API-Klasse in ihrer Gänze:

```
package ch.ost.gpu4java;

public final class GPU4Java
{
    public static void enterGPU()
    {
        // nop
    }

    public static void exitGPU()
    {
        // nop
    }

    private GPU4Java()
    {
        super();
    }
}
```

Eine Klasse, welche die obige API nutzt, könnte dann wie folgt aussehen:

```
import ch.ost.gpu4java.GPU4Java;

public class Example
{
    public static void gpuAdd(float[] a1, float[] a2, float[] a3)
    {
        GPU4Java.enterGPU();
        for(int i = 0; i < a3.length; ++i)
        {
            a3[i] = a1[i] + a2[i];
        }
        GPU4Java.exitGPU();
    }
}
```

Nachdem diese Klasse zu Java Bytecode kompiliert und anschliessend durch den Transpiler modifiziert wurde, würde sie sinngemäss so aussehen:

```
import ch.ost.gpu4java.GPU4Java;

public class Example
{
    static
    {
        System.loadLibrary("gpu4java");
    }

    static native void gpu4java$gpuAdd$0(float[] a1, float[] a2, float[]
a3);

    public static void gpuAdd(float[] a1, float[] a2, float[] a3)
    {
        gpu4java$gpuAdd$0(a1, a2, a3);
    }
}
```

Anmerkung: In Java Source Code dürfen Methodennamen keine Dollar-Zeichen enthalten, aber in Java Bytecode ist dies erlaubt, und für automatisch erzeugten Code sogar vorgesehen.

Zusätzlich zur modifizierten JAR-Datei würde eine Shared Library namens "gpu4java" generiert, welche die übersetzte Implementation der gpuAdd-Methode enthält, und zu Laufzeit von der Klasse in die JVM geladen würde.

2.6.2 Pipeline

Hier ist eine graphische Darstellung der Pipeline unseres Transpilers:

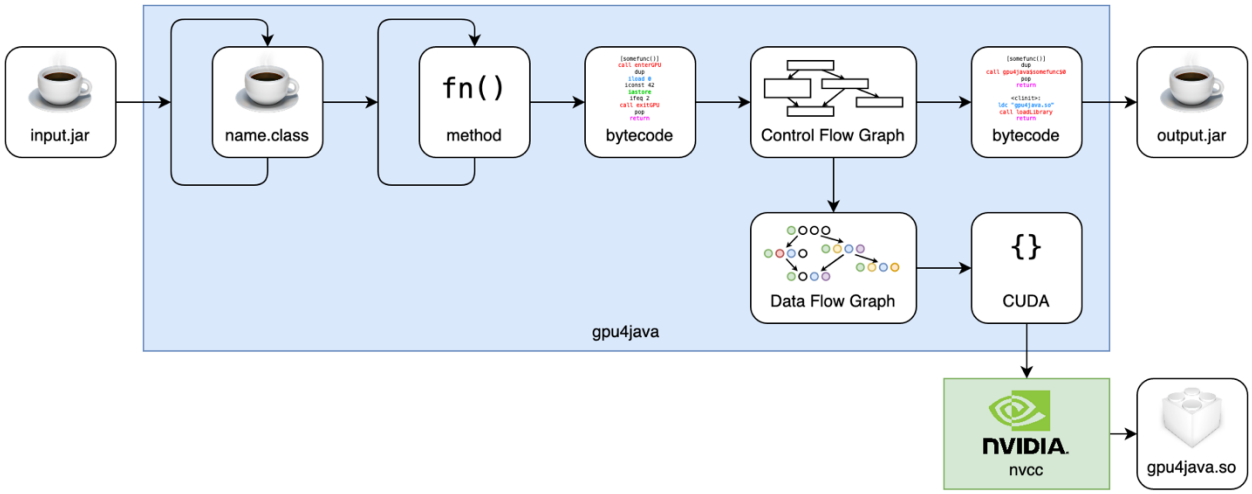

Abbildung 3 Pipeline

Die Pfade zu `input.jar` und `output.jar` werden als Kommandozeilen-Argumente an den Transpiler übergeben. Der Transpiler geht alle Dateien in der `input.jar` durch, lädt jede Datei mit `.class`-Endung in ein Byte-Array, und gibt dieses an ObjectWeb ASM weiter.

Das ASM-Framework parsed die Java-Klassen aus dem Byte-Array und gibt uns alle Fragmente davon per Visitor Pattern zurück. Das heisst also z. B., wir haben eine Implementation des `ClassVisitor`-Interfaces geschrieben, und für jede deklarierte Methode einer Klasse wird `visitMethod()` aufgerufen. Diese Methode gibt wiederum einen `MethodVisitor` zurück, auf welchem Methoden für alle Fragmente einer Methode aufgerufen werden, wie die Typen der Argumente, oder die individuellen Java-Bytecode Instruktionen, die die Implementation der Methode ausmachen. Innerhalb dieser Methoden können wir die Daten, die wir bekommen, beliebig verändern und dann an einen `ClassWriter` weitergeben, der die Fragmente wieder zu einer Java-Klasse zusammensetzt.

Da wir für unser Vorhaben jedoch Daten aus mehreren diese Methoden brauchen (insbesondere alle Instruktionen der Methode auf einmal), können wir die Daten nicht "inline" verändern, sondern müssen sie erst akkumulieren und am Ende der Methode schauen, ob und was wir damit machen wollen. Zu diesem Zweck haben wir einen `BufferingMethodVisitor` geschrieben, welcher alle relevanten Aspekte einer Methode abspeichert. Diese können dann als Ganzes bearbeitet werden.

2.6.3 Control Flow Graph

Sobald der `BufferingMethodVisitor` die Liste der JVM-Instruktionen einer Methode vollständig gesammelt hat, gehen wir diese durch und schauen, ob darin Aufrufe von `enterGPU` oder `exitGPU` vorkommen. Falls nein werden die ganzen gespeicherten Daten der Methode unverändert

zurückgegeben. Falls ja wird aus der Liste der Instruktionen ein Control Flow Graph erstellt, d. h. eine verlinkte Struktur aus Blöcken, welche nur Instruktionen beinhalten, die linear ausgeführt werden können. Diese Struktur hilft uns dabei, Veränderungen im Bytecode durchzuführen, und alle möglichen Code-Pfade zu bestimmen.

2.6.4 Data Flow Graph

Basierend auf dem Control Flow Graph wird dann der Data Flow Graph erstellt. Dabei werden mehrere Operationen gleichzeitig ausgeführt:

1. Start und Ende aller GPU-Abschnitte werden sich gemerkt, und es wird verifiziert, dass zu jedem `enterGPU` genau ein `exitGPU` existiert und umgekehrt, und dass die beiden Funktionen nicht mehrmals nacheinander aufgerufen werden. Prinzipiell wäre es denkbar, Code zu erlauben, welcher `enterGPU` oder `exitGPU` konditional aufruft, allerdings müsste der Transpiler in diesem Fall mathemtisch beweisen können, dass zu jedem enterGPU genau ein exitGPU aufgerufen wird. Aus Zeit- und Komplexitätsgründen wird dies nicht unterstützt.
2. Die Typen aller lokalen Variablen und aller Variablen auf dem Stack werden sich gemerkt. Dies erlaubt uns, die entsprechenden Methoden-Signaturen zu generieren, und die richtigen Instruktionen zu erzeugen, um Variablen an die native Methode zu übergeben.
3. Für alle Variablen-Zugriffe innerhalb der GPU-Abschnitte wird sich gemerkt, welche Variablen gelesen und geschrieben werden. Dadurch kann ermittelt werden, welche Variablen an die native Methode übergeben werden müssen, und welche diese zurückgeben soll. Hierbei ist speziell darauf zu achten, dass beim Schreiben einer Variable separate Listen geführt werden müssen, um zu bestimmen, ob eine Variable als Input oder Output markiert werden soll. Dies kann Anhand des folgenden Code-Beispiels erläutert werden:

```
int[] a = this.array;
int i = 1;
GPU4Java.enterGPU();
if(a.length > 1)
{
    i = 2;
}
a[0] = i; // (X)
GPU4Java.exitGPU();
System.out.println(i);
```

   Würde man nur eine Liste führen, um sich zu merken, welche Variablen überschrieben werden, so würde diese Liste an Stelle (X) entweder ja sagen, d. h. i wird innerhalb der GPU-Codes erzeugt und muss nicht als Input übergeben werden, oder nein, d. h. i wurde nicht verändert und ist kein Output. Beides ist falsch, weshalb zwei separate Listen notwendig sind.

Mit Hilfe aller obigen Daten werden letztendlich ein oder mehrere Teile des Control Flow Graphen "ausgeschnitten" und durch Code ersetzt, der alle notwendigen Variablen auf den Stack lädt, die neu generierte native Methode aufruft, und die zurückgegebenen Werte wieder so abspeichert, wie sie ohne

Transpiler-Eingriff vorhanden gewesen wären. Falls die Anzahl der im GPU-Abschnit modifizierten Variablen 0 oder 1 beträgt, ist der Fall trivial: der Return Type der native Method ist entweder `void`, oder der Type dieser einen Variable. Bei 2 oder mehr Variablen muss jedoch etwas Aufwand betrieben werden. Um jeden Fall richtig abdecken zu können, erhält die erzeugte Methode in diesem Fall den Retrun Type `Object[]`, und allfällige primitive Werte werden per "autoboxing" auf der einen Seite in Java-Objekte umgewandelt, und auf der anderen Seite wieder in primitive Typen extrahiert.

2.6.5 Generierung von C und CUDA

Der ausgeschnittene Teil des Control Flow Graphen wird dann zusammen mit gewissen Informationen aus dem Data Flow Graph an den CUDA Generator weitergegeben. Dieser schreibt darauf basierend einen CUDA-Kernel, indem einfach jede JVM-Instruktion nach der anderen in ein CUDA, bzw. C-Statement übersetzt wird. Dabei werden gewisse Operationen bewusst nicht unterstützt, so wie Zugriffe auf Felder oder Aufrufe von Methoden auf Java-Objekten. Ausserdem werden keine Operationen unterstützt, die neue Objekte oder Arrays erstellen. Diese lassen sich in GPU-Code schlichtweg nicht abbilden.

Des Weiteren wird hier die Implementation der native Methode erzeugt. Deren Aufgabe ist es, alle Argumente von Java-Typen in C-Typen umzuwandeln und entsprechend an die GPU anzubinden. Anschliessend wird der CUDA-Kernel aufgerufen, und danach alle benutzten Daten wieder zurück zu Java-Typen konvertiert.

Die meisten Typen lassen sich 1:1 ohne viel Aufwand abbilden, allerdings nicht Arrays. In C bestehen Arrays lediglich aus einem Pointer, in Java jedoch zusätzlich aus einer Länge. Zu diesem Zweck definieren wir den Typen `arr_t`, welcher einen Pointer und eine Länge enthält. Dies hat allerdings zur Folge, dass wir in einem ersten Schritt jedes Java-Array zu einem C-Pointer und einer Länge konvertieren müssen, in einem zweiten Schritt den Buffer auf GPU-Memory kopieren, und in einem dritten Schritt ein Objekt vom Typen `arr_t` erstellen und dieses ebenfalls auf GPU-Memory kopieren. Ähnlich aufwendig wird es beim Zurück-Kopieren, wobei hier noch geprüft werden muss, ob das Array innerhalb des GPU-Codes neu zugewiesen wurde. Zwar erlauben wir kein Erstellen von neuen Arrays, bestehende Arrays dürfen jedoch beliebig neu zugewiesen werden, und wenn zwei Array-Variabeln auf das gleiche Array zeigen, müssen wir das auch beim Zurück-Konvertieren zu Java-Objekten entsprechend abbilden.

Die ganzen erzeugten CUDA- und C-Funktionen werden als Strings erzeugt und gesammelt, einem "Prolog" versehen, welcher den JNI- und andere Header-Includes beinhaltet, und schliesslich wird das Ganze als eine Einheit an den Nvidia CUDA-Compiler übergeben und in eine Shared Library kompiliert.

2.6.6 Testing

Da unsere Pipeline fast ausschliesslich auf grossen und komplexen Java-Objekten arbeitet, wären Unit Tests in unserem Fall sehr umständlich gewesen, und wären ein Vielfaches grösser und komplexer ausgefallen als System-Tests. Aus diesem Grund haben wir uns in dieser Arbeit auf System-Tests beschränkt.

2.7 Limitationen

Unser Transpiler funktioniert grundsätzlich und kann beliebig komplexen Java-Code in CUDA-Code übersetzen. In der aktuellen Revision sind uns keine Fälle bekannt, in denen unser Transpiler falschen Code generiert. Allerdings gibt es eine grosse Anzahl an Features, die wir entweder bewusst nicht unterstützen, oder aus Zeit- oder Komplexitätsgründen weglassen mussten.

Allem voran steht die Parallelisierung: Aktuell läuft der von unserem Transpiler generierte Code nur in einem einzigen Thread, was natürlich keine Beschleunigung zur Folge hat. Leider mussten wir uns aus Zeitgründen darauf beschränken, überhaupt beliebigen Java-Code zu CUDA zu übersetzen. Mit ein paar Wochen mehr Zeit hätten wir die Parallelisierung allerdings wahrscheinlich erreichen können. Zwei Dinge haben uns dazu gefehlt: Einerseits hätten wir den Data Flow Graphen erweitern müssen um Code, der festetellen kann, welche Variablen in einer Schleife abhängig von der vorherigen Iteration sind, und andererseits hätten wir den CUDA Generator so anpassen müssen, dass Schleifen, deren Iterationen unabhängig voneinander sind, entsprechend eliminiert und deren Inhalt direkt als Threads auf der GPU ausgeführt werden.

Der zweite grosse Punkt sind Arrays. Da Arrays in Java "by reference" übergeben werden, können sie modifiziert werden, ohne Output-Variablen zu sein. Unser Data Flow Graph verfolgt aktuell nicht, ob Elemente von Arrays überschrieben werden, also müssen wir annehmen, dass alle Arrays beschrieben werden. Dies hat zur Folge, dass wir auch Arrays von der GPU zurück ins Host-Memory kopieren, die wir nicht müssten. Aus dem gleichen Grund werden aktuell nur eindimensionale Arrays unterstützt.

Ein ähnliches Problem liegt vor mit Output-Variablen im Allgemeinen. Wir stellen aktuell nur fest, ob Variablen innerhalb des GPU-Codes geschrieben werden, aber nicht, ob diese nach dem GPU-Code benutzt werden. Gibt es im GPU-Code also z. B. eine Schleife mit einer Zählvariable, so wird diese Zählvariable von unserem Transpiler als Output erkannt, auch wenn diese nachher nicht mehr benutzt wird.

Dann gibt es einige Instruktionen, die vom CUDA Generator nicht unterstützt werden, wie z. B. "dup2" oder "ldc". Diese wurden rein aus Zeitgründen nicht implementiert, und sollten mit je ca. 20 Zeilen Code implementierbar sein.

Auf der Ebene des Control Flow Graphen unterstützen wir allgemein keine Exceptions. D. h. Methoden, welche gpu4java nutzen, dürfen kein `throw`, `try`, `catch` oder `finally` beinhalten. Dies war eine bewusste Entscheidung, da diese den Control Flow Graphen sehr viel komplexer machen, und wir Exceptions innerhalb des GPU-Codes sowieso nicht sinnvoll unterstützten können.

Ähnlich verhält es sich mit impliziten Exceptions oder Errors wie NullPointerException, ArrayIndexOutOfBoundsException oder OutOfMemoryError. Wir haben uns dazu entschieden, in den Fällen, welche normalerwise solche Exceptions oder Errors erzeugen würden, das Programm stattdessen einfach zu beenden und eine Fehlermeldung auf der Konsole auszugeben.

2.8 Bedienungsanleitung

2.8.1 Dependencies

- Java 14 oder neuer
- Nvcc (Nvidia CUDA compiler)
- GNU make

2.8.2 Kompilierung

Mit einem einfachen "`make`" sollte das Projekt kompiliert werden können.

2.8.3 API

Die Klasse im api/ Ordner dient als API und exportiert zwei statische Methoden:

- `GPU4Java.enterGPU()`
- `GPU4Java.exitGPU()`

Dies ermöglicht es, Code zu schreiben wie folgt:

```
GPU4Java.enterGPU();
for(int i = 0; i < a3.length; ++i)
{
    a3[i] = a1[i] + a2[i];
}
GPU4Java.exitGPU();
```

2.8.4 Bedienung

Um einfach bedient werden zu können, stellen wir ein run.sh-Skript mit den nötigen Java CLI Argumenten zur Verfügung. Es kann wie folgt benutzt werden:

```
Usage: ./run.sh [-c] [-f] <in.jar> <out.jar>
Options:
    -c   Dump the C and CUDA code to stdout (intended for debugging).
    -f   Force overwriting of existing files.
```

Zusätzlich zu den obigen Argumenten und Flags gibt es einige Umgebungsvariablen, die den C/CUDA Compiler beeinflussen:

Variable	Standard-Wert	Effekt
NVCC	`nvcc`	Name oder Pfad Nvidia CUDA Compiler Programms.
NVCC_FLAGS	`–shared –x cu –Xcompiler –Wall –Xcompiler –O2 –Xcompiler –fPIC`	Standard-Argumente an C/CUDA Compiler. –x wird benötigt, da der Code stdin an der Compiler übergeben wird.
CFLAGS	*(nichts)*	Zusätzliche Argumente für den C/CUDA Compiler.
JAVA_HOME	*(nichts)*	Pfad des Java-Installationsverzeichnisses. Es wird erwartet, dass der JNI-Header im Pfad `$JAVA_HOME/include/jni.h` existiert.

Eine Shared Library wird im gleichen Ordner wie `out.jar` erstellt. Auf einigen Betriebssystemen muss man danach zum Ausführen der JAR-Datei ein weiteres Argument angeben, damit alles korrekt funktioniert. Dies kann wie folgt getan werden:

```
java –Djava.library.path=. –jar out.jar
```

2.9 Zeitplan / Meilensteine

Ziel	Deadline
Bytecode Parser (Find Annotation)	28.10.2021
Control Flow Graph (For-Loop)	04.11.2021
Data Flow Graph	11.11.2021
Bytecode Modification / Native function Call	18.11.2021
Java Bytecode to CUDA	25.11.2021
CUDA to shared Library	25.11.2021
Multiple For-Loops	09.12.2021
Optimisation	24.12.2021
Report	24.12.2021

2.10 Fazit

Das Projekt war anspruchsvoll, aber definitiv spannend und lehrreich. Wir trafen auf Hindernisse und kamen nicht immer voran wie geplant, sind jedoch mit dem Endresultat sehr zufrieden. Im Verlauf dieses Projektes haben wir erfahren, wie wichtig und hilfreich es sein kann, wenn man früh einen Architekturdurchstich hat und darauf aufbauen kann. Dies führt einen auf den richtigen Pfad von Anfang an und reduziert das Risiko, dass etwas technisch nicht machbar ist. Einen weiteren Punkt, den wir während dieser Arbeit gelernt haben, ist, dass Compiler und deren Optimierungen sehr komplex sind und viel Aufwand benötigen. Weiterhin haben wir gelernt, dass es sehr schwierig ist, Fortschritte an Projekten ohne graphische Komponenten für Drittpersonen ersichtlich zu machen.

3. Glossar

API	Application Programming Interface, Programmier-Schnittstelle
Bytecode	Binärer Programmcode, welcher von einem Programm ausgeführt bzw. interpretiert wird und nicht direkt auf dem Prozessor läuft
CFG	Control Flow Graph, Verlinkte Struktur, welche den Ablauf einer Funktion repräsentiert
CLI	Command Line Interface, Kommandozeile
CPU	Central Processing Unit, Prozessor
CUDA	Programmiersprache für Nvidia-Grafikkarten
DFG	Data Flow Graph, Information über die Abhängigkeiten von Daten innerhalb eines Programm-Teils
GPU	Graphical Processing Unit, Grafikkarte
JAR	Java Archive, Einheit bestehend aus mehreren kompilierten Java-Klassen
JNI	Java Native Interface, Java-Schnittstelle zu anderen Programmiersprachen
JVM	Java Virtual Machine, Laufzeitumgebung von Java
OpenCL	Open Computing Language, Generische Programmiersprache für Grafikkarten

4. Literaturverzeichnis

[1] **Vorlesung Luc Bläser**, Parallele Programmierung FS2021

Altimesh: https://developer.nvidia.com/blog/hybridizer-csharp/ (Stand 23.12.2021)

[2] **CUDA:** https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (Stand 23.12.2021)

Intel: https://www.intel.com/content/www/us/en/products/details/processors/core.html (Stand 23.12.2021)

5. Abbildungsverzeichnis

6. Anhang

6.1 Aufgabenstellung

A central processing unit (CPU) is designed to handle complex tasks, such as complex control flows, virtualization, memory management and multi-tasking. Modern multi-core architectures allow limited parallel execution. In contrast, graphical processing units (GPUs), offer only arith-metic operations but can run thousands of threads in parallel. Computation intensive software can be accelerated by offloading the calculation to GPUs. Software which has been written in C/C++ can, with moderate effort, be ported to CUDA and run on the GPU. However, software which runs on the JVM cannot benefit from these technologies since a full rewrite would be necessary. The goal of this project is to develop a framework which allows, e.g., a Java program on GPUs, with as little intervention from the programmer as possible. A possible approach which is followed by http://www.altimesh.com/ is as follows: 1. The programmer uses a special annotation to mark computation intensive loops 2. A tool, which has to be developed, reads the compiled bytecode and generates executable code, which can directly run on the GPU.