App-controlled LEGO 3-DoF robotic arm

Mobile Computing — winter term 2018/2019

Christoph Ulrich
HTWG Konstanz
Constance, Germany
christoph.ulrich@htwg-konstanz.de

Benjamin Schaefer
HTWG Konstanz
Constance, Germany
benjamin.schaefer@htwg-konstanz.de

***Zusammenfassung*—Die folgende Ausarbeitung beschreibt die Berechnung einer einfachen Wellenausbreitungsfunktion mithilfe des CUDA-Frameworks und die Darstellung der berechneten Daten mittels der Grafikbibliothek OpenGL. Ausgangssituation ist dabei eine stillliegende Wasseroberfläche. Simuliert wird die kreisförmige Ausbreitung einer Welle, die etwa von einem ins Wasser geworfenen Stein verursacht worden sein könnte. Die nach Evaluation ausgewählte Funktion zur Berechnung der Welle wurde zuerst zu Testzwecken auf der CPU implementiert und anschließend in einen CUDA-Kernel überführt. Danach wird ein weiterer Kernel konstruiert, mit dem die Oberfläche der errechneten Welle in Dreiecke aufgeteilt und dann mittels OpenGL gerendert wird. Anschließend werden die implementierten Kernel und unterschiedlichen Vorgehensweisen auf ihre Laufzeiten untersucht und verglichen. Zum Schluss erfolgt eine kritische Betrachtung des Projektes in Form von Optimierungsvorschlägen und eines Fazits.**

I. Planung und Ziele

Die Vorabplanung des Projektes sah die Simulation einer Wellenausbreitung (z.B. im Medium Wasser) ausgehend von einem Ausbreitungszentrum vor. Die Berechnungen sollten auf einem Grafikprozessor (im Folgenden kurz: GPU) mittels CUDA erfolgen und ein grauwertbildähnliches Format liefern, wobei die einzelnen Pixelwerte die Höhe der Welle an der jeweiligen Stelle repräsentieren sollten. Im Anschluss daran sollte aus den berechneten Daten ein Mesh generiert und dieses mittels OpenGL *gerendert* werden. Zusätzlich zu diesen Grundzielen wurden folgende optionale Ziele (Umsetzung nur bei vertretbarem Aufwand sowie vertretbarer Komplexität) vereinbart:

- Setzen des Ausbreitungszentrums per Mausklick
- Darstellung mehrerer Wellen mit unterschiedlichen Ausbreitungszentren und Wellenüberlagerung
- Rendern mit Textur

II. Vorarbeiten

A. Finden einer geeigneten Wellenfunktion

Zu Beginn galt es, eine Wellenfunktion zu finden, die den gewünschten Anforderungen entspricht, d.h. eine kreisförmige, mit der Zeit abschwächende und dreidimensionale Ausbreitung beschreibt. Am Ende der Recherche standen die nachfolgenden drei Ausbreitungsfunktionen zur Auswahl:

$$\eta(r,\theta,t) = \frac{\sqrt{gt}}{r^{\frac{5}{2}}} \cdot (\cos(\frac{gt^2}{4r}) - \sin(\frac{gt^2}{4r})) \tag{1}$$

r und θ beschreiben hier die Polarkoordinaten der Welle.

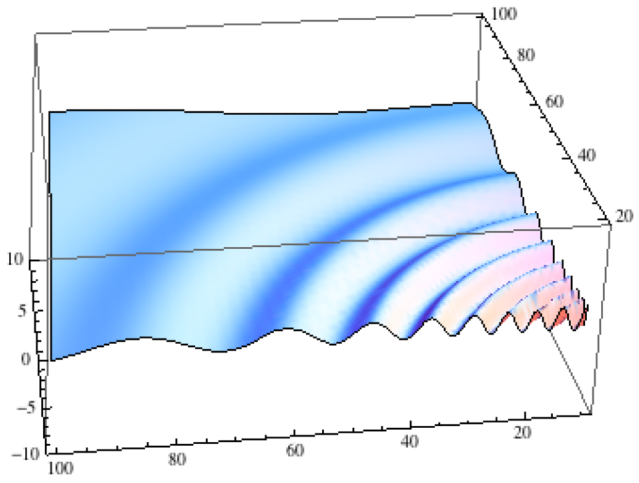

Fig. 1. 3D-Python-Plot zu Funktion (1)

$$-(\frac{1+\cos(12*\sqrt{x^2+y^2})}{\frac{1}{2}\cdot(x^2+y^2)+2}) \tag{2}$$

x und y beschreiben hier die kartesischen Koordinaten der Welle.

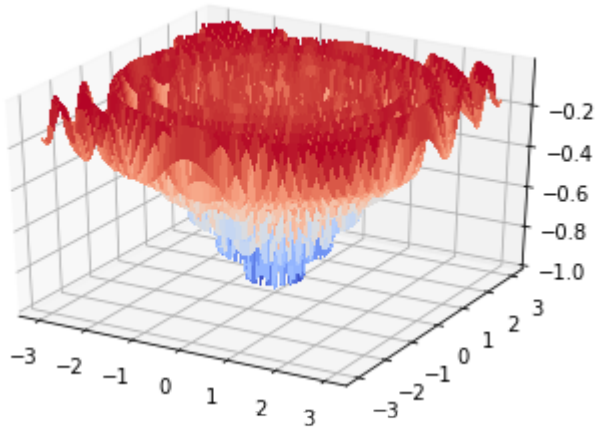

Fig. 2. 3D-Python-Plot zu Funktion (2)

$$\frac{\cos(\frac{1}{2} \cdot \sqrt{x^2 + y^2} - 6t)}{\frac{1}{2} \cdot (x^2 + y^2) + 1 + 2t} \tag{3}$$

x und y beschreiben hier die kartesischen Koordinaten der Welle.

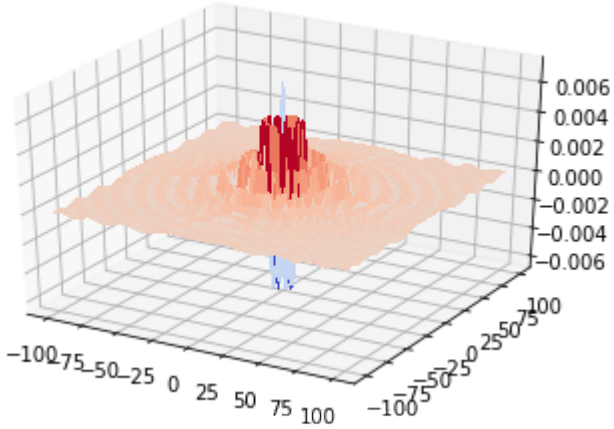

Fig. 3. 3D-Python-Plot zu Funktion (3)

Funktion (2), die sog. *drop-wave*-Funktion, berücksichtigt keine zeitliche Entwicklung, sondern berechnet die Wellenausbreitung lediglich zu einem schon fortgeschrittenen Zeitpunkt. Eine Möglichkeit hätte in der Anpassung der Funktion und dem Einbau eines zeitlichen Parameters bestanden. Aufgrund weiterer zur Verfügung stehender Alternativen wurde jedoch von diesem Vorgehen abgesehen und die Entscheidung fiel zugunsten von Funktion (3). Neben des relativ einfachen Aufbaus hat diese Ausbreitung den Vorteil eines deutlich sichtbaren Zentrums, was die reale Situation nach Eintauchen eines Objektes in eine Wassermasse hervorhebt.

B. Konzeptentwurf

Um von der reinen Formel zur schlussendlichen Visualisierung zu gelangen, wird die Vorgehensweise bei der Datenberechnung in zwei essentielle Schritte eingeteilt, die jeweils geeignete Datenstrukturen benötigen.

1) Höhenfeld: Die Ergebnisse der Berechnungen mit Funktion 3 repräsentieren die Höhe der Welle an der Position [x,y] zum Zeitschritt t, die in einem sog. Höhenfeld (engl.: *height field*) hinterlegt werden. Dieses Feld - ähnlich einem Grauwertbild - kann entweder als eindimensionale Liste oder als 2D-Grid gespeichert werden. Im ersten Fall wird auf den Wert an der Position [x,y] mit der Index-Umrechnung $y * size_y + x$ (typisches 2D-zu-1D-Mapping) zugegriffen. Das Höhenfeld muss zu jedem Zeitschritt t neu berechnet werden.

2) Dreiecke: Mit den Daten aus dem Höhenfeld sind zwar theoretisch alle Informationen zur Wellenstruktur vorhanden. Um die Welle jedoch später mit *OpenGL* darstellen zu können, müssen die errechneten Punkte aus dem Höhenfeld in Dreiecke transformiert werden.

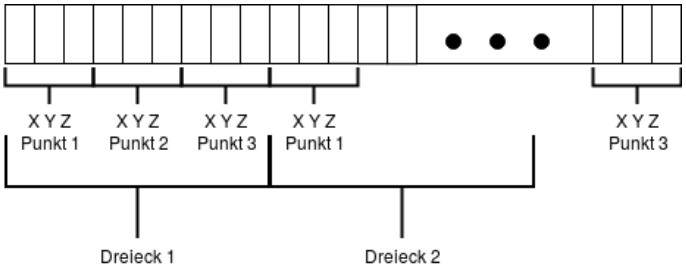

Fig. 4. Datenstruktur der Dreiecke

Die Dreieckskoordinaten werden nacheinander in einem Float-Array abgelegt, wobei ein Dreieckspunkt aus drei Werten (x ,y, z) besteht und daraus folgend ein gesamtes Dreieck aus neun Array-Einträgen besteht (siehe Abbildung II-B2)

III. Implementierung auf der CPU

Vor Umsetzung auf der GPU wurde die Berechnung der Wellenausbreitung mit klassisch sequentiellem Code auf der CPU implementiert. Damit war es auch möglich, im Voraus die allgemeine Funktionalität und die Korrektheit der Daten zu verifizieren. Dabei wurde mit zwei verschachtelten *for*-Schleifen über die gewünschte Anzahl von y- und x-Koordinaten iteriert und die Höhe der Welle an der jeweiligen Stelle berechnet. Um eine feinere Auflösung zu erreichen, wurde ein Skalierungsfaktor mit den Koordinaten verrechnet:

```
for (y=0; y<size_y; ++y) {
        for (x=0; x<size_x; ++x) {
                float fx=x*scale_x-trans;
                float fy=y*scale_y-trans;
                height_field[y,x] = ...;
```

Listing 1. Berechnung des Höhenfeldes auf der CPU - Pseudo-Code

Mit dem resultierenden Höhenfeld können nun die zur Darstellung notwendigen Dreiecke errechnet werden.

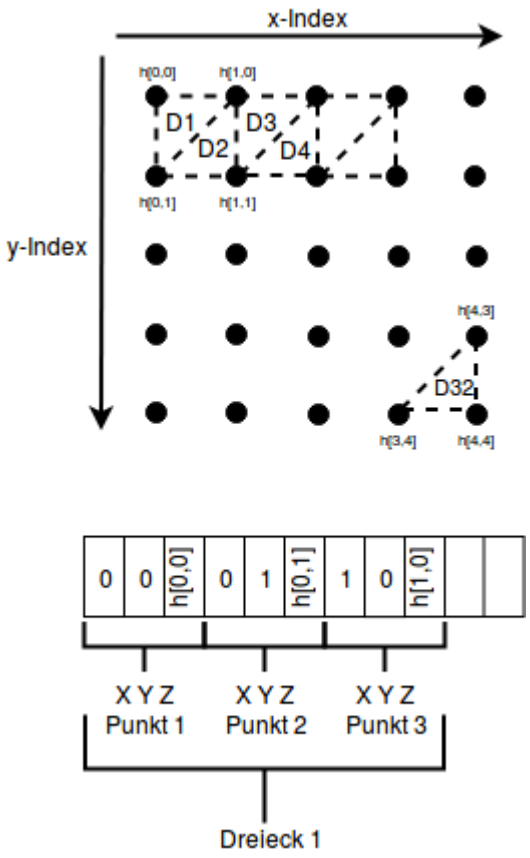

Fig. 5. Berechnung der Dreieckskoordinaten

Die Dreiecke werden nach dem in Abbildung III skizzierten Schema in ein eindimensionales Float-Array T geschrieben. x, y und $h[x, y]$ sind die Koordinaten eines zum jeweiligen Dreieck D_x gehörigen Punktes. Die Anzahl n_D der durch ein Höhenfeld h beschriebenen Dreiecke errechnet sich durch

$$n_D = (size_{h_x} - 1) \cdot (size_{y_x} - 1) \cdot 2 \tag{4}$$

Demnach beläuft sich die Größe des Arrays zur Speicherung der Dreieckskoordinaten T auf

$$size_T = n_D \cdot 3 \cdot 3 \tag{5}$$

Für das in Abbildung III gezeigte Beispiel gilt dann: $n_D = (5-1) * (5-1) * 2 = 32$ sowie $size_T = 32 * 9 = 288$ Die CPU-Code-Struktur zur Berechnung der Dreieckskoordinaten gleicht der zur Berechnung des Höhenfeldes:

```
for (int y=0; y<(size_y-1); ++y) {
        for (int x=0; x<(size_x-1); ++x) {
```

Listing 2. Berechnung der Dreieckskoordinaten auf der CPU - Pseudo-Code

IV. Implementierung auf der GPU

Im Folgenden ist die Analyse des CPU-Codes hinsichtlich Parallelisierbarkeit und die anschließende Portierung in parallel ausführbaren Code durch Konstruktion geeigneter CUDA-Kernel und Verwendung von Parallelisierungs- und Operation-Pattern beschrieben.

A. Analyse des CPU-Codes

Bei Untersuchung der relevanten Anteile des CPU-Codes lässt sich feststellen, dass innerhalb der zu parallelisierenden Aufgaben (Höhenfeld- und Dreiecksberechnung) keine Abhängigkeiten bestehen. Die Dreiecksberechnung benötigt jedoch als Input das Ergebnis einer abgeschlossenen Höhenfeldberechnung, d.h. diese Aufgaben werden sequentiell nacheinander ausgeführt. Da, wie in Listing 1. und Listing 2. zu sehen, über for-Schleifen diesselbe Rechenoperationen auf unterschiedliche Daten () angewendet werden, sind die einzelnen Jobs jeweils Daten-parallel und damit hervorragend geeignet für die Berechnung auf den *SIMD*-Cores der GPU. Im Hinblick auf die Granularität macht es deswegen Sinn, die Aufgaben jeweils in sehr kleine Teilaufgaben aufzuteilen, d.h. wenige Instruktionen pro Thread und dabei viele Threads zu verplanen.

B. Portierung nach CUDA

Nach erfolgter Analyse lässt sich die Strategie zur Parallelisierung nun ... Aufgrund der Feingranularität der Jobs ist das Ziel, für die Kernel die maximale Anzahl an Threads auszunutzen. Die beiden bei der Entwicklung eingesetzten Grafikkarten *GeForce GTX 940M* und *GeForce GTX 960M* sind hardwaretechnisch limitiert auf eine maximale Anzahl von 1024 Threads pro Block. Aufgrund einfacher Indizierung der 2D-Daten werden quadratische 2D-Blocks mit 32x32 Threads eingesetzt. Anhand der Größe der Wasseroberfläche wird die Dimensionierung des 2D-Block-Grids bestimmt:

```
dim3 block(32,32);
dim3 grid(height_field_size_x/block.x,
      height_field_size_y/block.y);
```

Listing 3. Bestimmung der Kernel-Parameter

In beiden implementierten Kernel werden die x- und y-Indizes zum Datenzugriff wie folgt aus den Kernel-Parametern bestimmt:

```
int x=(blockIdx.x*blockDim.x+threadIdx.x);
int y= blockIdx.y*blockDim.y+threadIdx.y);
```

Listing 4. Bestimmung der Indizes innerhalb der Kernel

In beiden Kernel ist ein Speichersicherheitsmechanismus vorgesehen, der dafür Sorge trägt, dass bei zu großer Anzahl an Threads im letzten Block kein Zugriff auf Speicherbereiche außerhalb des reservierten Bereichs der Eingangs- und Ausgabedaten stattfindet:

```
if((x*y)<
(height_field_size_y*height_field_size_x))
```

Listing 5. Speichersicherheit im Kernel garantieren

Wie in Abschnitt IV-A bereits angesprochen, wird im Kernel zur Berechnung des Höhenfeldes mit einer sehr feingranularen Struktur gearbeitet, d.h. jeder Thread führt nur wenige Instruktionen aus. Hier wird ein dem *Map*-Pattern ähnelndes Operation-Pattern eingesetzt. Das Map-Pattern verknüpft jeden

Thread mit einem Speicherelement, liest aus einem Datenelement und schreibt ein Datenelement. Das hier eingesetzte Pattern hat dagegen kein Quell-Datenelement - der Index selbst ist das Datum.
Der Kernel zur Berechnung der Dreieckskoordinaten ist etwas grobgranularer aufgebaut. Ein Thread erstellt aus einem ausgelesenen Datum aus dem Höhenfeld insgesamt 18 neue Einträge im Array zur Speicherung der Dreieckskoordinaten (siehe Abschnitt II-B2). Damit schreibt jeder Thread mehrere Datenelemente und arbeitet nach dem *Scatter*-Operation-Pattern.

V. Visualisierung der Wellenausbreitung mit OpenGL

Zum *Rendern* der Wasseroberfläche wurden im vorherigen Schritt Dreiecke generiert. Dreiecke sind in der Computergrafik die Standardgeometrie zum Darstellen von Oberflächen, da sie den Vorteil haben, immer planar zu sein und damit eine Fläche sehr gut approximieren zu können. OpenGL bietet alternativ andere geometrische Formen an, wie etwa Quads. Diese sind aber seit OpenGL 3.1 nicht mehr verfügbar und werden auch nicht zur Verwendung empfohlen. [1]

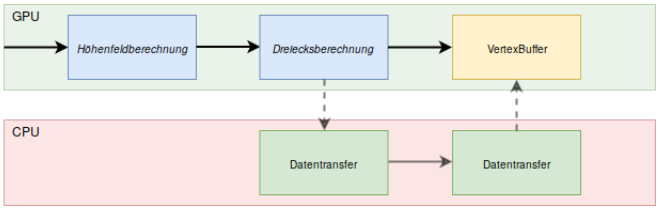

Fig. 6. Berechnungs- und Darstellungs-Workflow

Während der Umsetzung des Projektes wurden zwei unterschiedliche Formen des Datenflusses implementiert. Abbildung V zeigt die zwei unterschiedlichen Vorgehensweisen. Zu Beginn des Projektes wurde nach Berechnung der Dreieckskoordinaten mit dem Kopieren der Kernelresultate zurück zum Host ein unnötiger Umweg eingeschlagen. Im Laufe der Entwicklung hat sich herausgestellt, dass es wesentlich effizienter ist, die Kernelresultate im Speicher der GPU (im *VertexBuffer*) zu belassen. In CUDA kann über einen Pointer auf den Daten gearbeitet werden und OpenGL greift zum rendern der Daten auf den *VertexBuffer* zu.

A. Einfärbung der Wasseroberfläche

Die Einfärbung erfolgt anhand des Wertes der z-Koordinate. Ein *Vertex-Shader* übergibt einem *Fragment-Shader* die Knotenpunkte der Dreiecke. Je nach z-Wert wird dann die Einfärbung mittels der OpenGL-Funktion *mix()* realisiert. Diese Interpolationsfunktion erzeugt einen Gradienten von weiß bis blau und färbt die Oberfläche entsprechend ein.

VI. Auswertung

In diesem Abschnitt erfolgt eine Bewertung einzelner Testszenarien. Zu diesem Zwecke wurden an markanten Stellen Zeitstempel gespeichert, um mit mikrosekundengenauen Zeitdifferenzen zu überprüfen wie sich die verschiedenen Konfigurationen auf die Laufzeit auswirken. Im Folgenden werden die farblich kodierten Konfigurationen kurz erklärt.

1) Rot: Alle Berechnungen erfolgen auf der CPU, die Grafikkarte wird nicht verwendet.
2) Blau: Alle Berechnungen erfolgen auf der GPU, ein Rückschreiben des Ergebnissen in den Host-Speicher ist nicht erforderlich.
3) Grün: Alle Berechnungen erfolgen auf der GPU. Die Ergebnisse werden zurück zum Host kopiert, der damit das *Rendering* aktualisiert..

Die folgenden Abbildungen zeigen auf der y-Achse die über ca. 150.000 gemittelten Laufzeitwerte der jeweiligen auf der x-Achse abgetragenen Funktion in Mikrosekunden. Niedrigere Balken bedeuten daher bessere Performance. Der schwarze vertikale Strich innerhalb der Balken zeigt die Standardabweichung der jeweiligen Laufzeiten.

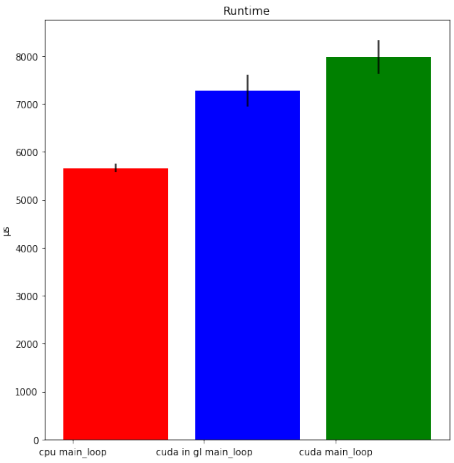

Fig. 7. Main Loop - 1 Thread pro Block

Die y-Achse von Abbildung VI zeigt die Laufzeit für einen Rendering-Schritt. In diesem Testszenario wurde ein Thread pro Block gewählt. Da Blöcke auf der Grafikkarte nicht parallel ausgeführt werden können, findet hier eine sequentielle Abarbeitung statt. Wie am grünen Balken zu erkennen, führt der dadurch entstehende Overhead (wie etwa Kopiervorgänge, Taktung, ...) zu einer deutlich höheren Laufzeit gegenüber der Ausführung auf einem CPU-Kern.

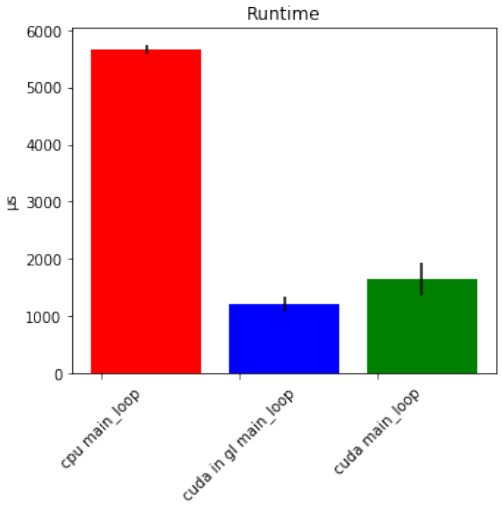

Fig. 8. Main Loop - 1024 Thread pro Block

Abbildung VI zeigt die Berechnung mit der maximal möglichen Anzahl an Threads pro Block. Dadurch reduzieren sich die Ausführungszeiten drastisch, da hier jetzt eine echt-parallele Verarbeitung stattfindet. Die CPU rechnet immer noch mit einem Kern, die Laufzeit bleibt daher unverändert (siehe Abbildung VI). Durch Einsparungen von Kopiervorgängen zwischen Host und Device ist die "cuda in gl"-Implementierung performanter.

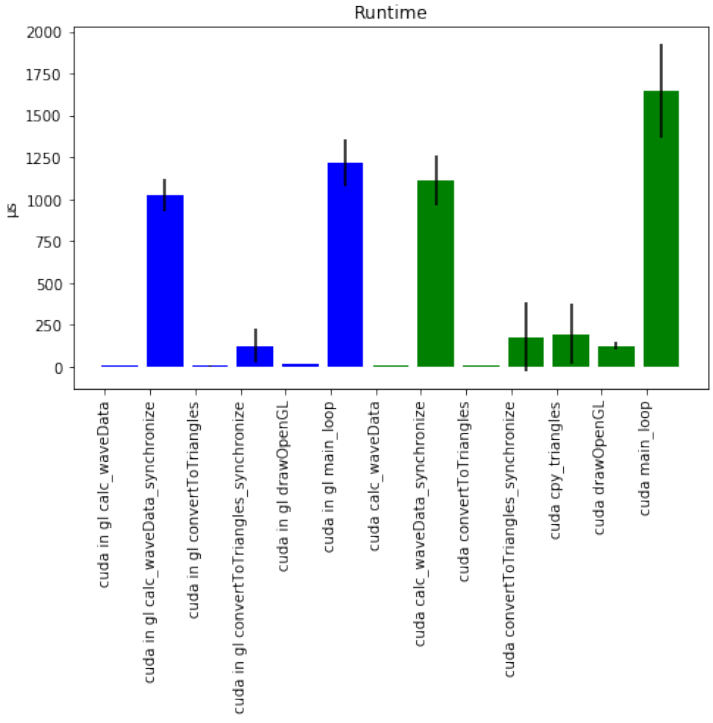

Fig. 9. CUDA vs. CUDA-GL

Abbildung VI stellt die zwei verschiedenen GPU-Implementierungen gegenuber. *calc waveData* repräsentiert hier den Kernel zur Berechnung des Höhenfeldes. Der linke Balken zeigt das Triggern des Kernels, der folgende zeigt die eigentliche Laufzeit des Kernels. Analog dazu initiiert *convertToTriangles* widerrum den Kernel zur Berechnung der Dreiecke. *drawOpenGL* stößt das *Rendering* an. Wird nicht nur auf der GPU gearbeitet (grün), müssen die errechneten Dreiecke mittels *copyTriangles* erst zurück zum Host und dann zum *Rendering* kopiert werden. Diese Kopiervorgänge entfallen bei der reinen GPU-Implementierung, bei der *drawOpenGL* deutlich schneller abläuft.

VII. Optimierung

Das Projekt kann technisch noch an einigen Stellen verbessert werden. Beispielsweise wird im Status Quo die maximale Anzahl an Threads pro Block manuell eingetragen. Dies kann durch Abfragen dieser Konstante über die *deviceQuery* automatisiert werden. Ebenso können, während CUDA-Kernel am rechnen sind, andere Operationen durchgeführt werden. Dies ist momentan ebenfalls nicht umgesetzt.
Eine mathematische Optimierung im Algorithmus kann ebenfalls vorgenommen werden: anstatt mit Funktion 3 das gesamte Höhenfeld mit allen Koordinaten zu berechnen, könnte auch ein einzelner Strahl im Raum berechnet und dieser dann mittels geeigneter Transformationen über die gesamte Fläche projiziert werden.

VIII. Fazit und Ausblick

Bei diesem Projekt hat sich deutlich gezeigt, wie viel Performance-Zugewinn durch Parallelisierung von Programmcode und Einsetzen der Grafikkarte zu erreichen ist. Vor allem bei Anwendungen in der Computergrafik, wo Instruktionen auf ein großes Datenset angewandt werden, ist die Effizienz spürbar. Voraussetzungen dafuer sind natürlich die in Abschnitt IV-B genannten Bedingungen für die Parallelisierung von Code. Besondere Aufmerksamkeit ist der Konfiguration der Kernelparameter zu widmen. Zu beachten ist auerdem, dass nicht durch zu großzügige Speicher-Allokationen der Zuwachs an Performance wieder zunichte gemacht wird.
Neben den im Abschnitt VII genannten Verbesserungen kann die Anwendung inhaltlich z.B. um die Darstellung multipler Ausbreitungszentren inkl. Wellenüberlagerung erweitert werden.

IX. Appendix

Ein erster Gedanke bei Beginn des Projektes zur Umsetzung der Datenberechnung war, die gewünschte Anzahl an Koordinatenpaaren (x,y) zu generieren, um diese dann zur Berechnung der Höhenwerte innerhalb eines CUDA-Kernels zu nutzen. Dazu wurde ein Mechanismus implementiert, der sich an den bereits in Python (numpy) oder Matlab integrierten *Meshgrid*-Funktionen orientiert und auf einen von Orange-OwlSolutions implementierten Kernel aufbaut.

A. Meshgrid-Kernel

Die entstandene Meshgrid-Funktion nimmt als Eingangsparameter Vektoren mit x- und y-Koordinaten und liefert dazu die entsprechenden x- und y-Koordinaten-Matrizen, die als Werte jeweils alle x- und y-Komponenten der gewünschten Koordinatenpaare enthalten:

```
x = [1, 2]
y = [1, 2, 3]
step = 1
[X, Y] = gpuMeshgrid(x, y, step)

X = 5x3

     1     2     3
     1     2     3
     1     2     3
     1     2     3
     1     2     3

Y = 5x3

     1     1     1
     2     2     2
     3     3     3
     4     4     4
     5     5     5
```

Listing 6. Meshgrid-Generierung - Pseudo-Code

Mit dem Parameter *step* können auch nicht-ganzzahlige Zwischenschritte erzeugt werden.
Die Voraberzeugung der Koordinatenpaare ist jedoch nicht notwendig. Wie in Abschnitt IV-B erläutert, werden die benötigten Koordinatenpaare direkt während der Berechnung des Höhenfeldes generiert. Die angepasste Funktion und der erweiterte Kernel zur Meshgrid-Erzeugung wurde aber trotzdem als nützliches Nebenprodukt des Projektes erachtet.

QUELLENVERZEICHNIS

[1] Khronos - OpenGL: Primitives - Triangle Primitives, https://www.khronos.org/opengl/wiki/Primitive